FreeBSD™ JOURNAL
July/August 2016

Amateur Radio a.n.d. FreeBSD. . .

- FreeBSD and RTEMS, UNIX in a Real-Time Operating System
- Tuning ZFS
- ScaleEngine and FreeBSD
- Getting the Job Done—The init System Debate Relaunched

Introducing the new
XG-2758 1U pfSense® Security Appliance

Fast 10 Gigabit networking at a price you can afford.

XG-2758 1U features include:

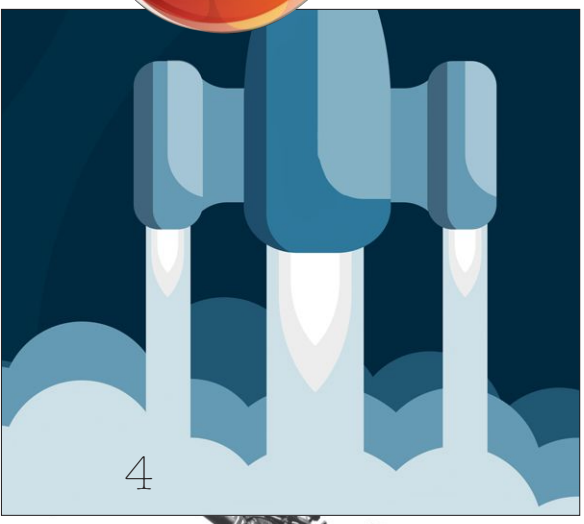
- 8 Core Intel® Atom™ C2758 2.4 GHz with AES-NI and Quick Assist Technology.
- 16GB ECC RAM
- 4x 1Gb Ethernet RJ45 via Intel i354 on-chip; 1 port configurable RJ45 or SFP.
- 2x 10Gb Ethernet SFP+ via Intel 82599 Niantic.
- Optional PCIe x8 slot available for further expansion.
- Preloaded with pfSense Open Source software. No maintenance, licensing or upgrade fees.
- Flexible Configuration - firewall, LAN or WAN router, VPN appliance, DHCP Server, DNS Server, multi-WAN and high availability.
- Fully extendable with add-on software packages such as Snort®, Squid, SquidGuard, Suricata, to enable IDS/IPS, load balancing, traffic optimization, reporting and monitoring.
- Create VPNs to the Amazon Cloud easily with our with Amazon® AWS™ Wizard.

pfSense®

Shop now at the official pfSense store or authorized partners worldwide.
http://store.pfsense.org/XG-2758

FreeBSD JOURNAL July/August 2016

Table of Contents



Columns & Reports

THE FREENAS MINI XL HAS ARRIVED

ENTERPRISE-CLASS HARDWARE, RUNNING THE WORLD'S MOST POPULAR OPEN SOURCE STORAGE OPERATING SYSTEM.

For more information on the FreeNAS Mini, visit **iXsystems.com/mini** today.

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Office Administrator • Michael Davis
davism@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation,
PO Box 20247, Boulder, CO 80308
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

Summer Reading

Here in the U.S. Northeast it is summer, and with summer comes summer reading. What better way to spend these days than reading about FreeBSD? This issue brings you several cool topics, including two pieces about ZFS, one on Tuning ZFS by Michael W. Lucas, the coauthor of two excellent books on the same topic, *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS*. As a convert to ZFS on my laptop where I use boot environments to reduce the risk of kernel development, I can say that learning more about this filesystem always improves my productivity. Michael's coauthor on the ZFS books, Allan Jude, has written an excellent article on the use of FreeBSD at ScaleEngine, a Content Distribution Network he has run since 2008. ScaleEngine uses FreeBSD, and specifically ZFS, to serve video and other content from a globally distributed set of data centers to clients all over the world.

Over the last couple years, many of us have watched as the Linux community tore itself apart over the systemd conflict. While schadenfreude is a fine thing, FreeBSD also needs to figure out a way to improve its own init system. Mark Heily has authored a piece on "Getting the Job Done" that will hopefully help FreeBSD to weather these waters with less Sturm und Drang.

While many readers of the *FreeBSD Journal* are familiar with FreeBSD as a whole system—from kernel to user space to tools, it has always been the case that various embedded operating systems, including the former Wind River's VxWorks, have adopted components of FreeBSD for their own purposes. The most commonly reused part of FreeBSD is the TCP/IP stack, one of the large and more complex parts of the system, and one that has been under active development since the original Berkeley Software Distributions in the 1980s. Several developers of RTEMS discuss their use of parts of FreeBSD in their own "Real-Time Executive for Multiprocessor Systems," which is an open-source, Real-Time Operating System (RTOS).

Batting cleanup for our main features is an article from Diane Bruce on Amateur Radio and FreeBSD. As Diane points out, the continuing drop in prices for computing power has brought this fascinating melding of the digital and analog worlds within the reach of nearly everyone who might have an interest in the topic.

As usual, our columnists keep you up-to-date on all things FreeBSD, including Steven Kreuzer with "svn update"; a Conference Report from Tim Moore; a profile on "New Faces of FreeBSD," as well as the Events Calendar, both from Dru Lavigne; and wrapping it all up, the latest from the FreeBSD Foundation, contributed by the Executive Director, Deb Goodkin.

Hopefully once you're done with our end-of-summer issue you'll be looking forward to the cooler climes and cool articles we'll have in September/October.

Sincerely, George Neville-Neil
For the *FreeBSD Journal* Editorial Board

SEE TEXT ONLY

GETTING THE JOB DONE

The init System Debate RE LAUNCHED

By Mark Heily

Over time, Unix operating systems have evolved a variety of mechanisms to launch programs and influence their execution. To put this into perspective, let's take a quick tour of the different kinds of programs that run on a typical FreeBSD system.

When the computer first boots up, the `rc(8)` mechanism launches programs called "services" that can perform one-time system initialization tasks, or execute daemons to run in the background. These services can also be stopped and started while the computer is running.

Every minute the computer is running, the `cron(8)` daemon wakes up and launches scheduled tasks. Cron also supports the ability to run `at(1)` jobs after a certain time interval has passed. It can also run `batch(1)` jobs when the system load average falls below a certain threshold.

There are many other mechanisms for launching programs. `Inetd(8)` launches programs when an incoming socket is created. `nice(1)` runs programs with a modified priority level. `chroot(8)` launches programs with an alternate root directory. `service(8)` launches programs with a "supervisor" process that can restart the program if it crashes. Jail managers like `jail(8)` launch programs within a jailed environment. Virtual machine managers like `iohyve(8)` launch programs that execute a

bhyve virtual machine. The `cloudabi-run(1)` command launches programs with a specific set of file descriptors pre-allocated.

Each of the mechanisms described above has its own configuration file format, its own particular style of usage, and essentially lives in its own isolated world. What if I told you there was a common underlying theme behind all of these different launchers? What if it were possible to have a single "launch all the things" mechanism that could become a common foundation for many of the operating system tools we enjoy today, as well as tools of the future?

I've started to explore these questions in greater detail, and have some interesting results to share. I've produced a working implementation of a new system called "the `jobd` job framework," or just "`jobd`" for short.

Overview of the Job Framework

`jobd` is a mechanism for launching and monitoring "jobs" that run within a single operating system instance. Jobs can be thought of as 1) some amount of work to be done, 2) the dependencies and preconditions before work can be started, 3) various ways to observe and interact with the program doing the work, and 4) a set of cleanup actions to perform after the work is complete.

That's quite a mouthful, so let's break it down into smaller chunks. We should start by going over the important concepts and common terms used in the job framework.

A job is defined by a JSON configuration file called a "manifest." The manifest contains a combination of a program, an execution context, properties, dependencies, and resources. These terms have specific meanings that are discussed below.

Within a job, the "program" is literally the path to the executable and the ARGV argument vector passed to the `execl(3)` system call. This begins the execution of a new program image, right after `jobd` calls `fork(2)` to create a new process.

The job's "execution context" represents all the changes to the child process that occur after the `fork()` call, but prior to the `exec()` call. Examples include: setting the user-id and group-id, calling `chroot(2)`, setting resource limits, setting the umask and nice value, and so forth.

System administrators need the flexibility to customize various aspects of how the job runs. For example, they might want to change the port number that a network daemon listens on. Job "properties" are the mechanism that allows for this control.

Job properties are simple key/value strings that can be substituted inside the manifest. Properties can also be queried directly, using library calls or the `jobcfg(1)` command. Taking it one step further, properties can be used to generate application-specific configuration files using a template; for example, the main `httpd.conf file` used by Apache.

Jobs can have "dependencies" that determine when the job should be stopped and started. Jobs can be started on-demand, on a fixed timer interval, at certain calendar dates/times, or when manually enabled by a system administrator. Jobs can also depend on the status of other jobs, so you can start and stop multiple jobs in a certain order.

Jobs can have "resources" which represent external things that are automatically created before the job starts, and are automatically destroyed when the job terminates. The job assumes ownership of the resources. This is most useful in combination with jails, as it allows you to create a jail on the fly to run a job.

We've covered the main aspects of what constitutes a "job" as far as the job framework is concerned. If you think back to the list of launchers in the first part of this article, there isn't a one-to-one mapping between what `jobd` offers and any of the existing mechanism. `Jobd` is very similar to `rc(8)` in that it supports all of the things you would need to manage services, but it isn't gravitationally bound to `init(1)` and the boot/shutdown process. It is more of an automaton, constantly running in the background and

Concept	Meaning	Example
Program	What to run	/usr/local/sbin/httpd
Context	How to run the program	setuid(2) to 'httpd'
Property	Information to tell the program	Listen on port 80
Method	A command the user can invoke to interact with the program	Run 'jobctl httpd status' to display server status information
Resource	Something the program needs before it runs	Install the 'httpd' package from the ports tree
Dependency	When the program should run	Whenever a client connects to port 80

reacting to events and conditions by starting, stopping, or restarting programs.

Now that the scope of the job framework is clear, it is worth mentioning a few things that are explicitly not in scope for the project. Unlike most of the init system replacements that have come out in the last few decades, `jobd(8)` is not trying to replace `init(8)` or usurp the role of `pid #1`. It will not handle early-boot tasks like setting up the console and `pttys`, or mounting filesystems listed in `/etc/fstab`. If you boot into single-user mode, I expect that this will still be handled by `init(8)` and `rc(8)` in much the same way as it is done now.

Despite the flexibility of the job framework to cover a wide range of use cases, I still would like to see it used first and foremost as a replacement for `rc(8)` and the current FreeBSD `init` system. This is a position that is likely to reawaken a lively debate among the community, as there are passionate defenders of the status quo. Nonetheless, it is a debate worth having again, in light of the potential benefits of adopting the job framework.

Some will ask, "Why change? Shell scripts have served us well since the beginning of BSD." I can't argue with the historical truth of that statement, but I will point out that the landscape of modern computing is far different from what it was during the early days of Unix. New problems demand new solutions.

Here are some reasons why I think that relying on a collection of shell scripts to power the init system is holding us back:

- The `rc(8)` system was not designed to run in the background and react to events and conditions that call for taking action. It runs once at boot time, and once at shutdown.
- Shell variables are simple pairs of strings that share a single global namespace. You can't build a complex hierarchical data structure out of simple key/value pairs. Without support for information-rich data structures, the `rc(8)` system will be limited to simple problem domains.
- Shell scripts mix data and code in the same file. This makes it impossible to programmatically edit `rc(8)` scripts, except for very trivial edits like changing the name of a variable.
- Shell scripts are imprecise, and unless they are written with a high level of paranoia, it is possible for their environment to be polluted by the person running the script.
- Managing large numbers of servers at scale requires the use of a configuration management system, such as Puppet or Chef. Teaching these systems how to configure individual services is very difficult, because each service has its own configuration file format and location, and these vary across platforms.
- The `rc(8)` system is not portable. Each variant of Unix has its own spin on the classic design, be it System V or BSD. Independent software vendors do not have the resources to support all of these different init systems, so they will target the most popular platforms such as commercial Linux distributions.

The job framework attempts to solve all of the problems listed above, or at least move us in the right direction.

History and Motivation

`jobd` grew out of an experiment to write a clone of the `launchd(8)` system found in MacOS X. Its command line syntax and fault-handling capabilities are heavily inspired by the Service Management Facility (SMF) system in Solaris. It is not merely a clone or mashup of these two Unix utilities; rather, it borrows liberally from the best features of each, while avoiding some of the less desirable aspects.

I was motivated to write `jobd` partly in response to the `systemd` debacle that caused a major schism in the Linux community. The adoption of `systemd` was a major contributor to my decision to switch my personal computer from Linux to PC-BSD. Having made the switch, I started looking around for little side projects that I could do to give back to the FreeBSD Project.

The desire to create a new init system was there, but it needed a spark to get things going. Coincidentally, around the time I switched to PC-BSD, the NextBSD project was announced, and after listening to a few of Jordan Hubbard's talks about the benefits of `launchd`, I was ready to help out.

Unfortunately, upon closer inspection, I did not agree with the technical approach that NextBSD used to port `launchd`; namely, the decision to write a partial implementation of the Mach microkernel as a compatibility layer, to essentially make FreeBSD pretend to be a half-baked variant of MacOS X.

I had previously spent years porting the `kevent(2)` API from FreeBSD to other operating systems, and knew firsthand the pain involved in

creating compatibility layers to make one kernel pretend to be a different kernel. This experience made me highly skeptical of the decision to create a Mach compatibility layer for `launchd`. Even if the Mach compatibility layer worked perfectly when added to the FreeBSD kernel, it would add a lot of technical debt that might not be acceptable to the FreeBSD Project. It would be difficult and time-consuming to port to other Unix-like operating systems, and there would not be much of an appetite for it.

I wanted something that worked "right now" on existing operating systems, using standard POSIX APIs where possible, so I started a new implementation of `launchd` from scratch, and called it "`relaunchd`."

Work Thus Far: Implementation

A relatively stable and feature-complete version of `relaunchd` was created and released as version 0.6. This contained most of the features found in the original `launchd`.

At an iXsystems hackathon, I worked closely with Kris Moore to try using `relaunchd` to manage a new PC-BSD service called SysAdm. We ran into several issues that pointed at the inadequacy of the original `launchd` design with respect to how software is packaged for FreeBSD. MacOS X does not have the concept of a ports tree, and software is usually installed via a graphical installer program that takes care of interacting with `launchd`. It turned out that human interaction with `launchd` leaves a good bit to be desired.

`Launchd` also does not have any support for user-defined properties, or exposing properties and methods to the user to manipulate how a service is launched. People coming from a background of using `rc(8)` would be understandably unhappy about losing the ability to define a `$foo_flags` variable in `rc.conf` to control the `foo` service, for example.

Another weakness of the `launchd` design is the lack of a fault management facility. If a daemon crashes, `launchd` will sit in an infinite loop trying to restart it. There is no built-in mechanism for determining that a service is faulty, and doing something about it. I have spent several years administering Solaris 10 servers, and like the idea that misbehaving services can be transitioned to a faulted state. In `jobd`, it will be possible to define a fault handler script that is executed whenever a

STATUS	LABEL
running	com.example.proprietary-agent
offline	org.freebsd.ports.apache24
disabled	org.freebsd.ports.mysql
running	org.freebsd.ports.postgresql
running	system.jail.my-jail-name
waiting	system.at.job-23
done	system.cron.job-4175
disabled	system.service.sshd
running	system.service.xorg
running	user.1001.kde-session

job fault is detected. This fault handler could do pretty much anything: send an email to an administrator, pop up a window on a desktop, send an alert to a monitoring system, and so on.

I started to be concerned at the mismatch between what I wanted **relaunchd** to do, and the original **launchd** design. This led me to take a step back and ask, "What are the problems that we are trying to solve, and is the classic **launchd** design an adequate solution?" As is often the case, the real world throws more problems at you than you originally anticipated.

This growing frustration with **launchd** led me to think about the concept of a job framework as a low-level operating system construct. This framework would provide a library and supporting command line tools to allow for building a variety of other operating system facilities, such as a service manager and a **cron(8)** replacement. I started to visualize splitting **relaunchd** in half; the lower layer dealing with starting and stopping jobs, and the upper layer presenting the face of a "service manager" to the world. Over time, other domain-specific front ends could be added, all of them sharing the same back-end job manager.

Coincidentally, other developers at the hackathon were working on the **iocage** jail manager tool, and the **iohyve** virtual machine manager. I started to look at the nascent "lower layer of **relaunchd**" as something that could be reused by **iocage** and **iohyve**, since fundamentally a jail and a virtual machine are just processes.

This idea of sharing code across disparate tools may sound crazy, but if you look closely, what do **rc**, **iocage**, and **iohyve** have in common?

* They start daemon processes, either automatically when the system boots, or manually when requested by an administrator.
* They allow these daemons to be stopped, started, enabled, and disabled.
* They allow system administrators to customize certain aspects of how the daemons are started, similar to the concept of "job properties."
* In the case of **iocage** and **iohyve**, they create virtual network interfaces for each process.
* They may require firewall modifications, such as adding NAT rules or filtering rules.

In addition to supporting a wider variety of use cases, I also wanted to avoid "feature creep" that would see **relaunchd** slowly assume new roles and responsibilities that would cause it to become a giant mess. In a layered design, it's important to have clear separation of concerns between the layers, and it was not clear where the line should be drawn between **relaunchd** and tools that are built on top of **relaunchd**. Even the name, "**relaunchd**," creates the impression that it is purely an init system in the style of **launchd(8)**. In order to be useful in a wider variety of situations, it needed to stop being an init system altogether.

To avoid the feature creep problem, the job framework provides a core set of features to be consumed by higher layers in the stack. These higher layers are expected to handle the details of job specialization and customization to fit their specific problem domain. For example, if **iocage** wanted to offer a "Jail Marketplace" where people could download pre-configured jails, all of this functionality could be implemented within **iocage** itself, and not require any modifications to the underlying job framework.

Once the overall picture of what a job framework should look like was in place, the decision was made to rename the project to "**jobd**" and make a clean break with the past. The existing **launchd(8)** daemon would be renamed to **jobd(8)**, and the problematic **launchctl(8)** utility would be retired and replaced with a new set of command-line tools that would provide a better user experience for developers, packagers,

and system administrators.

At this point, I didn't totally want to reinvent the wheel when designing the new `command-line` interfaces, so I borrowed some of the concepts from the Solaris(TM) SMF framework, and created three new CLI tools: `jobadm(1)`, which operates against the job database; `jobctl(1)`, which controls a single job, handing operational requests like starting and stopping the job; and `jobcfg(1)`, which would handle everything related to getting and setting job properties.

Owing to its simple design, `relaunchd` was a few thousand lines of C code. With the new mission of `jobd`, I started to be concerned about the potential challenges of implementing the new job framework functionality in standard C. The original `launchd` API was simple enough to implement in C, but it would be hard to implement all of the powerful new functionality in C. This led to the decision to rewrite the project in C++ to take advantage of the richer set of data structures and move to a more object-oriented design.

Moving to C++ posed some initial challenges, not least of which was my total lack of practical experience with the language. After some initial growing pains, the benefits of C++ started to come to light, and it became easier to implement powerful functionality that would have been difficult to do in standard C. Learning a new language is also fun and interesting, and raises the challenge level of whatever you are trying to accomplish.

Goals

The overall goals of the `jobd` project are:

- Explore the idea of job management at the operating-system level. This includes educating people about job management concepts, and getting developers to think about how to solve problems using jobs.
- Be the best job management framework on the market; provide a general purpose framework for launching jobs within a single operating system instance; handle all aspects of the job lifecycle: setup, startup, supervise, kill, teardown. Be the reference implementation of the job management API.
- Make the easy things easy, and the hard things possible; make it trivial to start processes in a consistent manner. The hard thing is getting firewall rules, jails, datasets, network interfaces, configuration files, package dependencies, etc., all playing nice; jobd will make it possible to tie all of these things together with a single configura-

tion file.

- Unify disparate mechanisms; serve as a foundation for the development of new operating-system mechanisms, such as replacements for `rc(8)`, `cron(8)`, and `jail(8)`.
- Be portable to a wide variety of POSIX-compliant operating systems.
- Peacefully coexist with legacy systems, but drive innovation and standardization forward.

How to Help

`Jobd` is still a young project, and there is plenty of opportunity for interested people to participate and help shape the direction of the system.

For FreeBSD developers, this is an opportunity to help make `jobd` a system that works best on FreeBSD, and to showcase some of the compelling features of FreeBSD, such as jails, Capsicum, and ZFS. In return, `jobd` will make FreeBSD a better operating system by modernizing the init system, improving boot times through parallelism and on-demand execution, providing a centralized job database, with a consistent set of tools that reduce code duplication.

Speaking of jails, one of the more interesting features of `jobd` that needs to be built is a robust sandboxing mechanism. Building a sandbox can be hard work, but `jobd` will make it easy by leveraging the available security features of each platform to automatically set up the sandbox for each job.

Eventually, I would like `jobd` to offer a plugin API so that third-party developers can build extensions that solve problems I haven't even imagined. Exposing a plugin API would also allow `jobd` to live in the base system, while various ports could add plugins to extend the base functionality. For example, if there was a demand to have greater integration with D-Bus, someone could write a plugin for this, and that would not cause jobd to depend on D-Bus.

Another exciting development opportunity is to work on the IPC mechanism. The job framework uses a slightly modified version of JSON-RPC over a Unix-domain socket. So far, this is only used for communication between different binaries in the framework. I would like to open this up as a general purpose IPC framework for jobs, so that one job can invoke a remote method provided by another job, with `jobd` acting as an on-demand mediator to spin up the target job as needed. Having a general-purpose structured local IPC mechanism integrated with `jobd` opens up a world of new design approaches for building distributed applications that run within a single operating system.

If you are interested in developing jobd, the best way to start is to contact Mark Heily (mark@heily.com) and request access to the Slack channel. There is also a mailing list; see the GitHub site for details. Before diving into the code, it would be good to have a conversation about what your interests and skills are, and then we can figure out some potential areas to collaborate on.

Even if you are not a developer, there are plenty of ways to help out. We need people to test `jobd`, ask questions, write documentation, spread the word on social media and other forums, review proposed features, and more.

Lastly, I would urge FreeBSD developers and ports maintainers to experiment with adding `jobd` support to their existing software.

Closing Remarks

"Something old, something new, something borrowed, and something blue."—*Traditional English bridal rhyme*

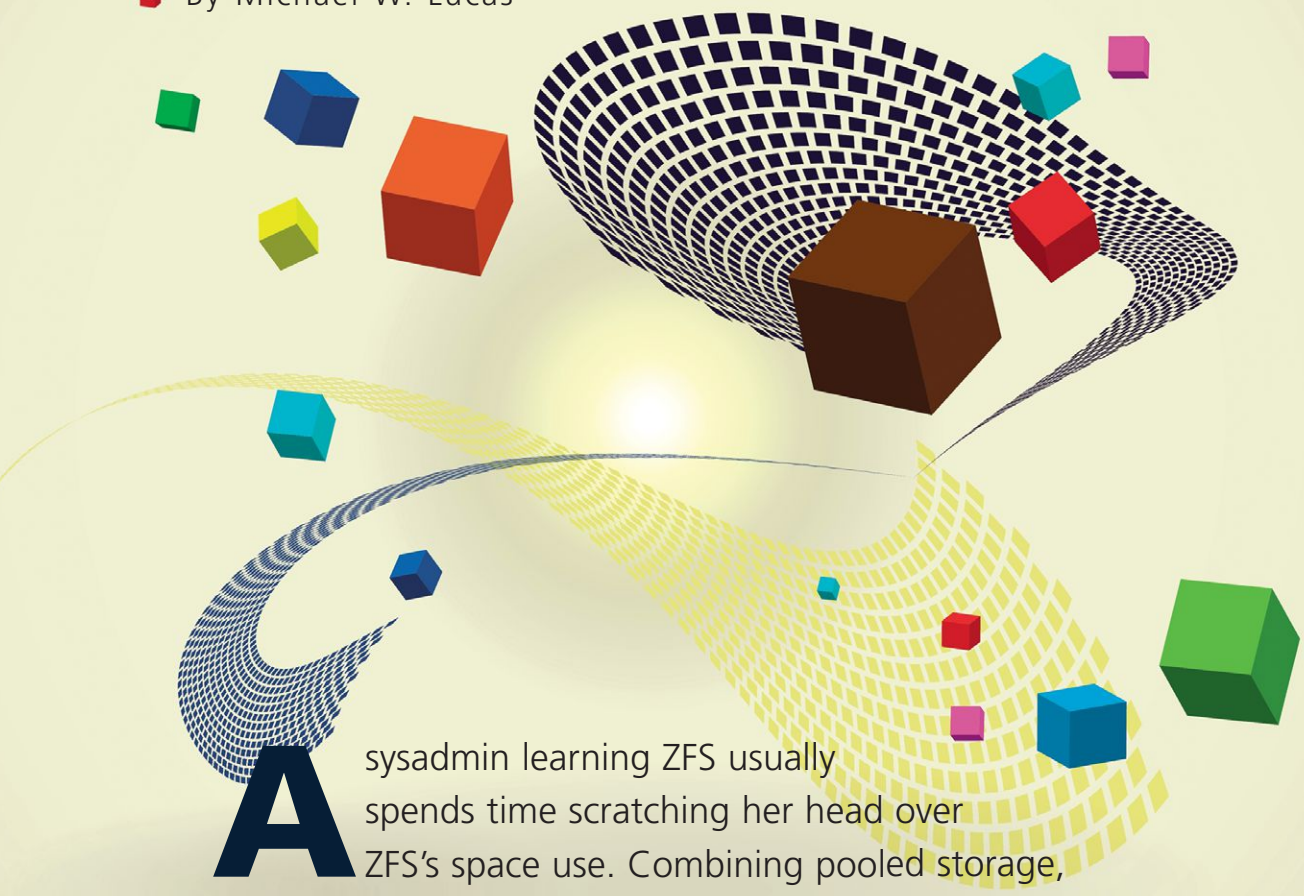
`Jobd` attempts to inhabit the "Goldilocks zone" of software design. It is not too traditional, which would limit its usefulness compared to existing tools. It is not too modern, which would make it difficult for users to learn and hinder a smooth transition. It borrows the best ideas of the init systems that have come before it, while avoiding the worst aspects of previous systems. It is innovative in the sense of combining existing concepts in a novel way, and above all else it should be practical and useful to getting the job done and solving real problems.

MARK HEILY is a professional systems engineer with a DevOps background in operating system deployment, automation, and tool development. His most successful open-source project to date has been `libkqueue`, a portable implementation of the FreeBSD `kqueue(2)` API. He also developed a portable version of the Grand Central Dispatch mechanism of MacOS X. He enjoys playing musical instruments, drinking craft beer, tinkering with computers, and currently lives in Raleigh, North Carolina.

SEE TEXT ONLY

Tuning ZFS

By Michael W. Lucas



A sysadmin learning ZFS usually spends time scratching her head over ZFS's space use. Combining pooled storage, datasets, snapshots, and clones makes ZFS space utilization very complicated. When you start mucking with the `recordsize` property for datasets, space utilization can swerve straight into the Twilight Zone.

The `recordsize` property gives the maximum size of a logical block in a ZFS filesystem dataset. The default `recordsize` is 128 KB, which comes to 32 sectors on a disk with 4 KB sectors, or 256 sectors on a disk with 512 byte sectors. The maximum record size was increased to 1 MB with the introduction of the `large_blocks` feature flag in 2015. Many database engines prefer smaller blocks, such as 4 KB or 8 KB. It makes sense to change the `recordsize` on datasets dedicated to such files. Even if you don't change the `recordsize`, ZFS automatically sizes records as needed. Writing a 16 KB file should take up only 16 KB of space (plus metadata and redundancy space), not waste an entire 128 KB record.

This comes most into play with databases.

Databases and ZFS

Many ZFS features are highly advantageous for databases. Every DBA wants fast, easy, and efficient replication, snapshots, clones, tunable caches, and pooled storage. While ZFS is designed as a general-purpose filesystem, you can tune it to make your databases fly.

Databases usually consist of more than one type of file, and since each has different characteristics and usage patterns, each requires different tuning. We'll discuss MySQL and PostgreSQL in particular, but the principles apply to any database software.

The most important tuning you can perform for a database is the dataset block size—through the `recordsize` property. The ZFS `recordsize` for any file that might be overwritten needs to match the block size used by the application.

Tuning the block size also avoids write amplification. Write amplification happens when changing a small amount of data requires writing a large amount of data. Suppose you must change 8 KB in the middle of a 128 KB block. ZFS must read the 128 KB, modify 8 KB somewhere in it, calculate a new checksum, and write the new 128 KB block. ZFS is a copy-on-write filesystem, so it would wind up writing a whole new 128 KB block just to change that 8 KB. You don't want that. Now multiply this by the number of writes your database makes. Write amplification eviscerates performance.

While this sort of optimization isn't necessary for many of us, for a high-performance system it might be invaluable. It can also affect the life of SSDs and other flash-based storage that can handle a limited volume of writes over their lifetimes. Of course the different database engines don't make this easy, and each has different needs. Journals, binary replication logs, error and query logs, and other miscellaneous files also require different tuning.

Before creating a dataset with a small `recordsize`, be sure you understand the interaction between VDEV type and space utilization. In some situations, disks with the smaller 512-byte sector size can provide better storage efficiency. It is entirely possible you may be better off with a separate pool specifically for your database, with the main pool for your other files.

For high-performance systems, use mirrors rather than any type of RAID-Z. Yes, for resiliency you probably want RAID-Z. Hard choices are what makes systems administration fun!

All Databases

Enabling lz4 compression on a database can, unintuitively, decrease latency. Compressed data can be read more quickly from the physical media, as there is less to read, which can result in shorter transfer times. With lz4's early abort feature, the worst case is only a few milliseconds slower than opting out of compression, but the benefits are usually quite significant. This is why ZFS uses lz4 compression for all of its own metadata and for the L2ARC.

When the Compressed ARC feature lands in OpenZFS, enabling compression on the dataset will also allow more data to be kept in the ARC, the fastest ZFS cache. In a production case study done by Delphix, a database server with 768 GB of RAM went from using more than 90% of its memory to cache a database to using only 446 GB to cache 1.2 TB of compressed data. Compressing the in-memory cache resulted in a significant performance improvement. As the machine could not support any more RAM, compression was the only way to improve.

ZFS metadata can also affect databases. When a database is rapidly changing, writing out two or three copies of the metadata for each change can take up a significant number of the available IOPS of the backing storage. Normally, the quantity of metadata is relatively small compared to the

default 128 KB record size. Databases work better with small record sizes, though. Keeping three copies of the metadata can cause as much disk activity, or more, than writing actual data to the pool.

Newer versions of OpenZFS also contain a `redundant_metadata` property, which defaults to ***all***. This is the original behavior from previous versions of ZFS. However, this property can also be set to ***most***, which causes ZFS to reduce the number of copies of some types of metadata that it keeps.

Depending on your needs and workload, allowing the database engine to manage caching might be better. ZFS defaults to caching much or all of the data from your database in the ARC, while the database engine keeps its own cache, resulting in wasteful double caching. Setting the `primarycache` property to ***metadata*** rather than the default all tells ZFS to avoid caching actual data in the ARC. The `secondarycache` property similarly controls the L2ARC.

Depending on the access pattern and the database engine, ZFS may already be more efficient. Use a tool like `zfsmon` from the `zfs-tools` package to monitor the ARC cache hit ratio and compare it to that of the database's internal cache.

Once the Compressed ARC feature is available, it might be wise to consider reducing the size of the database's internal cache, and instead letting ZFS handle the caching. The ARC might be able to fit significantly more data in the same amount of RAM than your database can.

MySQL–InnoDB/XtraDB

InnoDB became the default storage engine in MySQL 5.5 and has significantly different characteristics than the previously used MyISAM engine. Percona's XtraDB, also used by MariaDB, is similar to InnoDB. Both InnoDB and XtraDB use a 16 KB block size, so the ZFS dataset that contains the actual data files should have its recordsize property set to match. We also recommend using MySQL's `innodb_one_file_per_table` setting to keep the InnoDB data for each table in a separate file rather than grouping it all into a single `ibdata` file. This makes snapshots more useful and allows more selective restoration or rollback.

Store different types of files on different datasets. The data files need 16 KB block size, lz4 compression, and reduced metadata. You might see performance gains from caching only metadata, but this also disables `prefetch`. Experiment and see how your environment behaves.

```
# zfs create -o recordsize=16k -o compress=lz4 -o
redundant_metadata=most -o primarycache=metadata mypool/var/db/mysql
```

The primary MySQL logs compress best with `gzip`, and don't need caching in memory.

```
# zfs create -o compress=gzip1 -o primarycache=none mysql/var/log/mysql
```

The replication log works best with `lz4` compression.

```
# zfs create -o compress=lz4 mypool/var/log/mysql/replication
```

Tell MySQL to use these datasets with these `/usr/local/etc/my.cnf` settings.

```
data_path=/var/db/mysql
log_path=/var/log/mysql
binlog_path=/var/log/mysql/replication
```

You can now initialize your database and start loading data.

MySQL – MyISAM

Many MySQL applications still use the older MyISAM storage engine, either because of its simplicity or just because they have not been converted to using InnoDB.

MyISAM uses an 8 KB block size. The dataset record size should be set to match. The dataset layout should otherwise be the same as for InnoDB.

PostgreSQL

ZFS can support very large and fast PostgreSQL systems if tuned properly. Don't initialize your database until you've created the needed datasets.

PostgreSQL defaults to using 8 KB storage blocks for everything. If you change PostgreSQL's block size you must change the dataset size to match.

On a default FreeBSD install PostgreSQL goes in `/usr/local/pgsql/data`. For a big install, you probably have a separate pool for that data. Here I'm using the pool `pgsql` for PostgreSQL.

```
# zfs set mountpoint=/usr/local/pgsql pgsql
# zfs create pgsql/data
```

Now we have a chicken-and-egg problem. PostgreSQL's database initialization routine expects to create its own directory tree, but we want particular subdirectories to have their own datasets. The easiest way to do this is to let PostgreSQL initialize and then create datasets and move the files.

```
# /usr/local/etc/rc.d/postgresql oneinitdb
```

The initialization routine creates databases, views, schemas, configuration files, and all the other components of a high-end database. Now you can create datasets for the special parts.

PostgreSQL stores databases in `/usr/local/pgsql/data/base`. The Write Ahead Log, or *WAL*, lives in `/usr/local/pgsql/data/pg_xlog`. Move both of these out of the way.

```
# cd /usr/local/pgsql/data
# mv base base-old
# mv pg_xlog pg_xlog-old
```

Both of these use an 8 KB block size and you would want to snapshot them separately, so create a dataset for each. As with MySQL, tell the ARC to cache only the metadata. Also tell these datasets to bias throughput over latency with the `logbias` property.

```
# zfs create -o recordsize=8k -o redundant_metadata=most -o primarycache=metadata logbias=throughput pgsql/data/pg_xlog
# zfs create -o recordsize=8k -o redundant_metadata=most -o primarycache=metadata logbias=throughput pgsql/data/base
```

Copy the contents of the original directories into the new datasets.

```
# cp -Rp base-old/* base
# cp -Rp pg_xlog-old/* pg_xlog
```

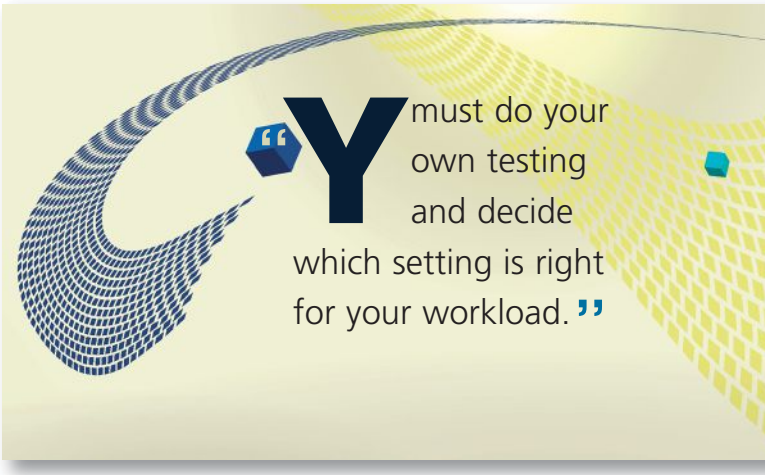
You can now start PostgreSQL.

Tuning for File Size

ZFS is designed to be a good general-purpose filesystem. If you have a ZFS system serving as file server for a typical office, you don't really have to tune for file size. If you know what size of files you're going to have, though, you can make changes to improve performance.

Small Files

When creating many small files at high speed in a system without a SLOG, ZFS spends a significant amount of time waiting for the files and metadata to finish flushing to stable storage.



If you are willing to risk the loss of any new files created in the last five seconds (or more if your `vfs.zfs.txg.timeout` is higher), setting the sync property to *disabled* tells ZFS to treat all writes as asynchronous. Even if an application asks that it not be told that the write is complete until the file is safe, ZFS returns immediately and writes the file along with the next regularly scheduled `txg`.

A high-speed SLOG lets you store those tiny files both synchronously and quickly.

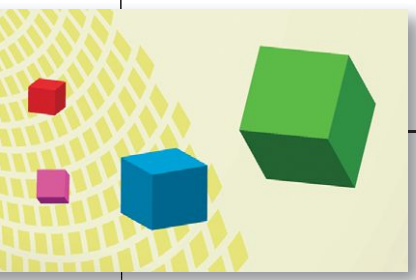
Big Files

ZFS recently added support for blocks larger than 128 KB via the `large_block` feature. If you're storing many large files, certainly consider this. The default maximum block size is 1 MB.

Theoretically, you can use block sizes larger than 1 MB. Very few systems have extensively tested this, however, and the interaction with the kernel memory allocation subsystem has not been tested under prolonged use. You can try really large record sizes, but be sure to file a bug report when everything goes sideways. The sysctl `vfs.zfs.max_recordsize` controls the maximum block size.

Once you activate `large_blocks` (or any other feature), the pool can no longer be used by hosts that do not support the feature. Deactivate the feature by destroying any datasets that have ever had their `recordsize` set to larger than 128 KB.

Storage systems struggle to balance latency and throughput. ZFS uses the `logbias` property to decide which way it should lean. ZFS uses a `logbias` of *latency* by default, so that data is quickly synched to disk, allowing databases and other applications to continue working. When dealing with large files, changing the `logbias` property to *throughput* might result in better performance. You must do your own testing and decide which setting is right for your workload. ●



MICHAEL W. LUCAS is the author of *Absolute FreeBSD, Absolute OpenBSD,* and *DNSSEC Mastery,* among others. He lives in Detroit, Michigan, with his wife and a whole mess of rats. Visit his website at https://www.michaelwlucas.com.

SEE TEXT ONLY

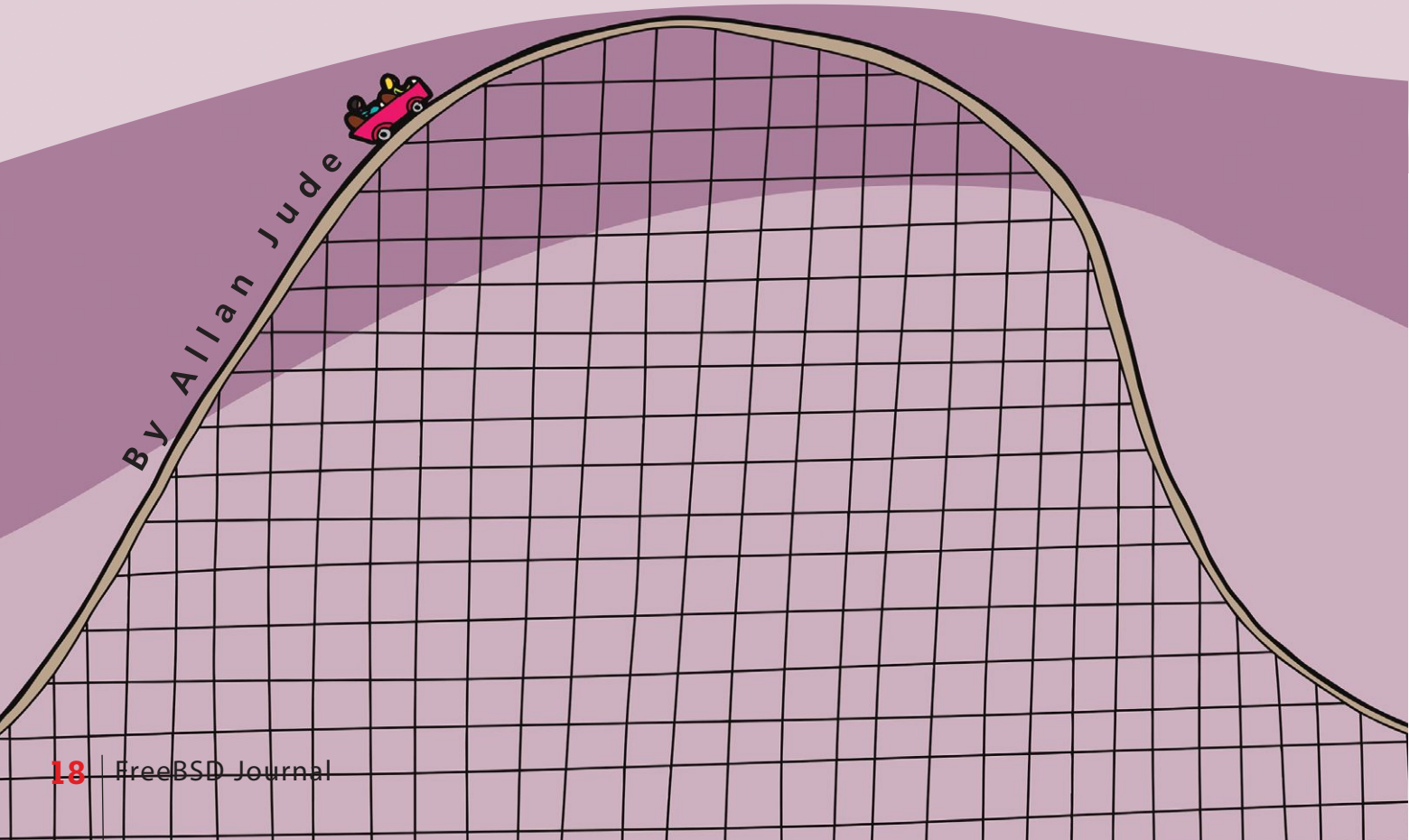
ScaleEngine and FreeBSD

When ScaleEngine was founded, it was based on FreeBSD because that is what the principals were most familiar with. Throughout the past eight years, the decision to use FreeBSD has served us well and provided us with distinct competitive advantages.

How ScaleEngine Started

ScaleEngine started its CDN (Content Distribution Network) entirely by accident. Originally, ScaleEngine was an auto-scaling application environment designed to host forums, large web apps, and other high-complexity high-traffic sites. After a short time, we quickly started to push too much bandwidth out of our primary colocation facility. In working around this problem, we quickly grew into a CDN and eventually pivoted our business to focus on that market.

Bandwidth in most colocation, and in Internet transit in general, is measured at "95th percentile." The basic concept is that the amount of traffic pushed to the Internet is measured every 5 minutes throughout the month. At the end of the month, the measurements are sorted, and the top 5% are discarded. The peak amount of usage that remains determines your bill. Usually there are two prices involved. The customer has a commitment level—a minimum amount of bandwidth they buy each month—at a fixed price.



By Allan Jude

Then there is a burst or overage price. The customer is able to use more than the committed amount of bandwidth, but if their 95th percentile exceeds the commitment, they must pay for the additional usage at the higher burst price. This has some benefit to both sides. For the transit providers, it ensures that customers pay for the amount of bandwidth they sustain during peak times, which ensures the transit provider can keep that amount of capacity available and encourages customers to commit to a larger amount of bandwidth to avoid the higher overage rate. For the customers, it offers them the flexibility to use more bandwidth when needed while only having to pay for the committed rate. The customer also has the ability to use the entire capacity of the connection for up to 5% of the month (approximately 36 hours), without any additional cost. The busiest hour of each day or the busiest 90 minutes of each weekday do not count against your final bandwidth bill.

ScaleEngine was in a position where we did not want to increase our bandwidth commitment, but also needed to avoid paying expensive overage fees each month. The majority of the traffic that was consuming the bandwidth was large image files we hosted for various sites, including a screenshot-sharing application. To offset this, we rented a server on the east and west coast of the U.S. Bandwidth for these servers was priced differently, based on total monthly volume, rather than peak usage. We directed traffic for static content—mostly the larger images—to these servers, which received the content via rsync from our primary site. This reduced the bandwidth consumption of our primary colocation to well within our commitment, since the majority of the traffic was now the plain text content of the sites we hosted. Demand for faster-loading images saw us add similar servers in Europe.

Dummynet: Smart Traffic Shaping

From the customer side, the downside to 95th percentile billing is that if your peak time during a month exceeds that free 5%, then you pay for that peak amount of traffic as if you had used that rate for the entire month. This means that there is often an advantage to smoothing out spikes in your usage. This is where Dummynet comes in. Dummynet is the traffic-shaping feature of IPFW, FreeBSD's native firewall. Using a pair of pipes and some queues, we were able to rate-limit replication traffic between our servers such that a large number of new files would no longer create a spike in traffic. Instead of being replicated at full speed, saturating our connection, and creating a large spike, replication would be rate-limited by a pipe and would yield to higher-priority customer traffic in the queue. This flattening of the spike saved thousands of dollars while only causing a minor replication delay.

For more on IPFW and Dummynet, see the May/June 2014 edition of the *FreeBSD Journal*.

Reliability and Flexibility

When you have to manage more than 100 servers, it helps to have a stable and reliable OS. With our server lifecycle we have a mix of FreeBSD versions, 9.3, 10.3, and -CURRENT. Having painless upgrades inside a stable branch (10.2 to 10.3) is a huge benefit, and we can quickly upgrade to the latest version without worrying about fallout in our application stack.

FreeBSD's approach to third-party libraries and applications has been an important part of our ability to deploy new features. The FreeBSD ports system is a rolling release, meaning that new applications and updated versions of those applications are added daily. The ports system also allows us to not be at the mercy of the vendor-provided compile-time options. We use `poudriere(8)` to compile our own versions of the set of packages we use, with the options that suit us. However, if we need something else, especially just temporarily, we can fall back to the packages provided by FreeBSD. The fact that the same ports tree works on all three supported branches of FreeBSD means we can have the latest version of nginx on every server,

be it 9.3 or the latest -CURRENT. Being able to get the latest version of nginx rather than what was the latest version when FreeBSD 9.0 was released, means we can rapidly adopt new upstream features. The FreeBSD ports system provides another important feature: multiple versions of the same application. We can choose between PHP 5.6 and PHP 7.0, or nginx and nginx-devel. A quarterly stable branch of the ports tree is available for those who wish to avoid the daily churn.

How FreeBSD and ZFS Made the Difference

When we started selling CDN services directly rather than as part of a hosting service, the architecture needed an overhaul. We had a number of issues that needed addressing, but FreeBSD provided a solution to each one of them.

We migrated away from rsync, which was too slow once the number of files grew beyond a few tens of thousands, and we needed to be smarter about which content we cached on each different edge server to avoid wasting precious storage space. We also needed to avoid wasting bandwidth replicating data that was rarely requested after it was a few weeks old. Again, the goal was to reduce the bandwidth pressure from our primary storage servers, to avoid having to buy additional expensive Internet transit. We switched to the nginx caching module, which saves files to the edge server the first time they are requested. This worked well at first, using hash-based directories and filenames to break the large number of files into manageably sized directories. Eventually we encountered performance problems with UFS, the default FreeBSD filesystem. The "dirhash," a cache of directory metadata, was limited by default to only a few megabytes of memory. Even when the size of the cache was expanded and the time-to-live increased, walking through the directories was painfully slow, which adversely affected cold startup times. Switching to ZFS changed everything.

ZFS has a much smarter caching system. Most filesystems use a standard LRU (Least Recently Used) cache, which, when full, removes the item used the longest time ago to make room for a new item. ZFS, instead, uses the ARC (Adaptive Replacement Cache), which consists of four lists. The first, the MRU (Most Recently Used), is very similar to LRU. There is also a separate MFU (Most Frequently Used) cache for files used most often. This provides an important optimization: when walking through the entire directory structure of cached objects, which can cause the entire LRU/MRU to be cycled, all of the items that were in the cache are removed and replaced by the entries being walked. Now the cache is full of items we likely will not use again, and we have to wait for the cache to recover to provide a reasonable performance boost. With the addition of the MFU, the files we use most frequently are not purged from the cache by the directory walk. Now we also had control over how much of the filesystem cache could be used for metadata. Unlike UFS, ZFS will only purge items from the metadata cache if it is full or if the memory is needed elsewhere. ZFS limits the metadata to 25% of the total cache size by default, but this can be adjusted if needed. In one of our use cases, hosting the thumbnail images for a very popular related content plugin, we were storing more than 30 million small files. The files were stored on an array of SSDs, so throughput was not an issue. ZFS allowed us to cache only the metadata in ram so that the entire directory structure would reside in ram, and only reading actual data blocks would result in reads going to the SSDs. This greatly lowered latency on cold files, compared to caching a mix of data and metadata for hot files and nothing for cold ones.

ZFS also brought with it a number of other options and features we could use and provide to our customers. One of the first lessons we quickly learned was to create a dataset for each customer. When we had just one dataset with a directory for each customer's video files, with a full regimen of snapshots, if a customer canceled their account we couldn't recoup the space without removing the snapshots. This would cause us to lose the history for every other customer in the process. Once we transitioned to a dataset per customer, we could remove snapshots or the entire filesystem for a customer and get the space back immediately. This also allowed us to create reservations and quotas for customers, to ensure they always had enough space, or to limit the amount of space they used.

ZFS also brought with it the final nail in rsync's coffin, block-level replication. Now we can replicate entire filesystems to our edge servers as an atomic operation. Doing an incremental update

only takes the time required to transfer the actual blocks that have been added or updated; there is no "scan" time or directory walking involved. When dealing with 30 million small files, the time savings over rsync are astronomical.

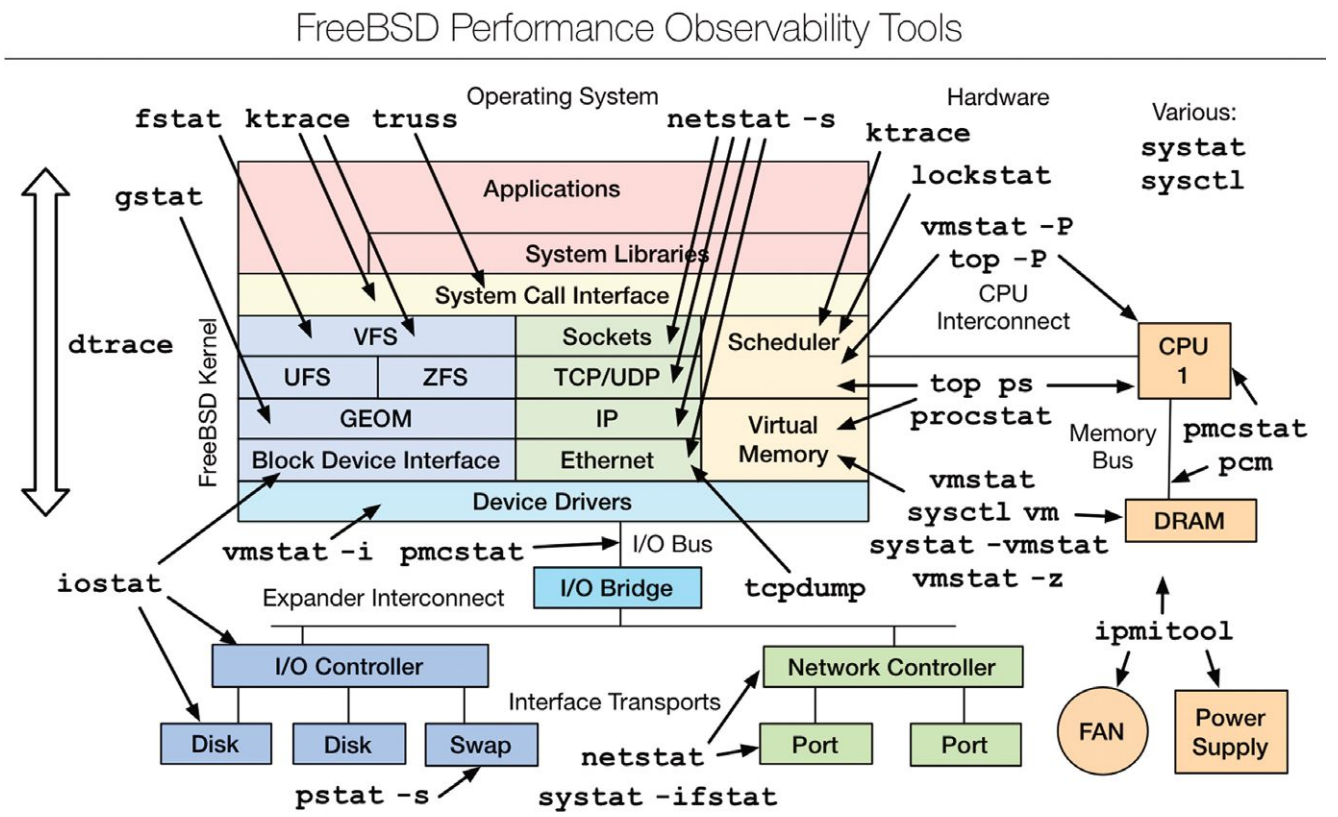
ScaleEngine hosts the package repository for PC-BSD using ZFS replication. A dataset was created on one of our storage servers and the PC-BSD team uploads the latest packages into a new directory. As they are uploading, our server takes snapshots every 15 minutes and replicates those to an array of edge servers. Once the upload is complete, the directory is moved in place of the old repository and the edges atomically update. Combined with our Global Server Load Balancer (GSLB), which stops routing traffic to an edge if its replication is delayed, this strategy has worked exceedingly well. Now that ZFS has support for resuming interrupted replication and the PC-BSD project has upgraded the connection of its build servers, we will start experimenting with having PC-BSD push a ZFS dataset directly to us rather than transferring the files individually.

Observability

Any good system administrator should be like a petulant 5-year-old asking "why" in a recursive loop. FreeBSD provides the tooling required to answer such questions, which goes a long way to being able to solve problems. The `top(1)` command in FreeBSD is very powerful while at the same time not consuming a large amount of resources itself, which would limit its usefulness for measuring the load on the system. It also has an I/O mode which can answer the question: which process(es) are causing all of this I/O? Then `fstat(1)` can tell which files that process has open. For a broader view, `gstat(8)` shows the I/O broken down by GEOM, which corresponds to disks and partitions. If throughput is not as high as expected, look at the number of IOPS being consumed. Does the device have any IOPS remaining with which to service your requests? The average latency measure can tell you if a disk is struggling to keep up with the load. A higher than expected latency under modest load can suggest a failing disk.

The excellent diagram below by Brendan Gregg (http://www.brendangregg.com/Perf/freebsd_observability_tools.png) illustrates the various parts of a FreeBSD system, and which tools can be used to observe them.

Dtrace has been invaluable in helping us track down issues in our application stack and follow them through the kernel. Whether you want to



watch the congestion window of a specific TCP connection or generate a graph of the average write latency to your disk array, it is just a matter of a few lines of D code. We used a series of DTrace probes to monitor a server during a local ZFS replication–duplication 10-TB of customer data to a second dataset. By looking at how long it took to flush all dirty data to the pool, the "sync latency," we were able to optimize the settings and achieve a 25% performance gain. By adjusting the maximum amount of dirty data, the dirty data threshold (the buffer is nearing full, so a sync is started), and the transaction timeout, we were able to delay the sync cycle (which suspends reads to reduce write latency) to only happen once 24 GB of data was waiting to be written, which would take around 5 seconds to write. The default tuning meant that a write happened every time 4 GB of data was dirty, causing the pool to frequently switch between reading and writing.

Networking

Advancements in the FreeBSD network stack have been extremely helpful to us. The pluggable TCP congestion-control algorithm system allowed us to experiment with different algorithms and determine that HTCP provided the best performance for trans-Atlantic ZFS replication. In cases like ours, often called long, fat networks, fluctuations in latency are not an indicator of congestion, and minor packet loss is not unexpected. HTCP is more aggressive than the default New Reno algorithm and gets up to speed faster and recovers from packet loss more quickly.

Support for modifying the default, initial-congestion window (RFC6928) was required to allow us to remain competitive with the small-file performance of the large competing CDNs. This setting limits the number of segments of data that can be sent before receiving acknowledgment from the other side. The initial value before the experimental RFC was between 2 and 4 segments and was changed to 10. This greatly reduced the load time for small HTTP objects, as the total number of round-trips would drop by 4. Increasing the size of the initial congestion window also allows for faster recovery in the case of packet loss during the early stages of the connection. This can greatly improve performance when the cause of the packet loss is not congestion, especially in the case of wireless connections.

With FreeBSD 11, the entire network stack has become pluggable, in such a way that different applications can use a different network stack. Theoretically this would allow us to use one set of optimizations for traffic from our edges to end users, and another for traffic over our backhaul network. This could become especially important because of the different types of traffic we serve. Flash video streams are a single long-lived connection at a constant modest bitrate. To provide the best experience, the focus should be on consistency and avoiding latency and packet loss. HLS video streams are a constant series of medium-sized (~1 MB) HTTP requests. Maximum burst performance is the most desirable outcome; however, if the device is mobile, via 4G or WiFi, packet loss can be expected, but does not necessarily indicate congestion. HTTP Progressive streaming, is similar to a regular download, but done as a series of HTTP Range Requests. This method is typically used by mobile devices, desktop browsers, HTML5 players, and set-top boxes. In this case, the optimal strategy is to avoid congestion while sending the data as quickly as possible. In the final case, HTTP Downloads, like the weekly episodes of the BSDNow.tv podcast, or PC-BSD packages, the goal is to avoid contention with the more latency- and congestion-sensitive streams coming from the same server, while offering the best possible speed.

Final Thoughts

To summarize why we choose FreeBSD to power our business: the world's most respected networking stack combined with the most reliable filesystem ever developed were brought together as an actively developed but extremely stable operating system all under a liberal, copyfree license. It is no wonder more bits are pushed by FreeBSD than anything else. ●

ALLAN JUDE is VP of Operations at ScaleEngine Inc., a global HTTP and Video Streaming CDN, where he makes extensive use of ZFS and FreeBSD. He is also the host of the video podcasts BSDNow.tv (with Kris Moore) and TechSNAP.tv. He is a FreeBSD src and doc committer, and was elected to the FreeBSD Core team in the summer of 2016. Allan is the coauthor of *FreeBSD Mastery: ZFS* and *FreeBSD Mastery: Advanced ZFS* with Michael W. Lucas.

Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

SEE TEXT ONLY

FreeBSD and RTEMS, UNIX in A REAL-TIME OPERATING SYSTEM

By Chris Johns, Joel Sherrill, Ben Gras, Sebastian Huber, Gedare Bloom

RTEMS developers have a long history of using FreeBSD code in the RTEMS source base and the results have been very good. RTEMS is not alone in doing this, and the challenges and issues RTEMS faces using FreeBSD source code are the same that others face.

RTEMS is a POSIX real-time operating system, or RTOS, and like all open-source projects has limited resources. All available effort needs to be focused on the real-time parts of the operating system. RTEMS looks to use quality, suitably licensed source code where possible—a combination of code from Newlib, RTEMS, and the FreeBSD TCP/IP stack—to provide a surprisingly robust subset of POSIX.

What follows is a summary of the history, what has been accomplished over the last 20 years, and how integration problems were solved while minimizing the maintenance burden.

RTEMS

RTEMS stands for Real-Time Executive for Multiprocessor Systems and has been open-source software since 1991. RTEMS has been used in some significant applications over the last 25 years and the project has a healthy community. It was developed by On-Line Applications Research (OAR) in Huntsville, Alabama, under contract from the U.S. Army in response to program managers watching missiles head down a test range and explode—each bundled with an operating system license.

RTEMS started life as a traditional embedded RTOS kernel typically found in the late 1980s and early 1990s. The user API was based around the VMEBus Industry Trade Association (VITA) Real-Time Executive Interface Definition (RTEID) 2.1 standard. The kernel's C source code was cross-compiled to a library, and a single standalone static executable was created by linking the application's object files with the RTEMS library. Embedding usually involved burning the binary image into an EPROM using some type of external programming tool.

An application and the kernel shared the same address space, and there was no memory management or virtual memory. An RTEMS application was statically linked with the OS and operated in the same single address space. The single-address-space environment was like a single process with no protection and a 1:1 mapping for the virtual addresses to the physical address space. The first versions of RTEMS had no C Library or filesystem, no networking, and device drivers and hardware support were often custom for the specific hardware and considered part of the application.

RTEMS has supported multiple processors for 20 years. The multiprocessor support is based around separate address spaces running on separate processors interconnected using a bus architecture and message passing. Applications can access and manage resources distributed across a number of nodes.

RTEMS has evolved and grown and is now much more sophisticated. However, some of these fundamental aspects of embedded, real-time systems remain and are valid.

Today RTEMS has symmetric multiprocessing (SMP) support. In some respects, the easy part is implementing SMP support in the OS because designing deterministic, real-time SMP applications is even more challenging. The problems in the OS are solved once, but each application faces similar challenges related to race-condition bugs, resource contention, and effective utilization of multiple cores while having predictable, correct, and safe behavior.

Real-time

Today, real-time software and real-time operating systems are as important as ever. The development of applications for them is alive and well, and so is the development of the operating systems to support these applications. The increase in computing power in smaller and smaller devices has brought powerful, server-grade operating systems such as FreeBSD to small, embedded devices, and these devices can have the performance to meet a range of real-time applications. However, these systems are still not fully deterministic, and the ability to complete a full schedulability analysis is problematic. RTEMS's performance can be deterministic, and it provides a rich suite of scheduling algorithms to meet the demands of real-time applications.

RTEMS has extended its support for high-performance, real-time computing by adding SMP for multi-core processors. Real-time software on SMP hardware is a difficult and complex challenge. To aid the application architects, RTEMS provides a powerful set of configuration interfaces. Cores can be grouped and assigned a specific scheduling algorithm, and threads can be assigned to a core affinity set and therefore associated with a specific scheduling algorithm. The ability to partition the components of a real-time design lets the architects manage the difficult task of schedulability analysis. RTEMS lets interrupts be assigned to specific cores letting the user manage latency when under load.

Kernel Architecture and APIs

The RTEMS Kernel source can be found in the git://git.rtems.org/rtems.git repository. It has four major components:

1. SuperCore
2. Application Programming Interfaces
3. Services
4. Board Support Packages (BPS)

SuperCore

The SuperCore is a super set of functionality exported to users via the various application programming interfaces or APIs. The SuperCore contains all the important, real-time algorithms in RTEMS. It contains the scheduling, locking, synchronization, clock, interrupt and SMP support. It also contains the architecture and CPU support for context switching and low-level interrupt management for each of the supported architectures.

The resources created and managed by the various user interfaces can coexist because everything maps to a SuperCore resource. This means a thread created in one user interface can block on a mutex created by a different user interface. This is a powerful feature, as software components can be implemented by different user interfaces and can be combined into a single executable. The support extends to the scheduling and time domain. This means the run-time profile of an application is independent of the API used to write it.

Application Programming Interfaces

RTEMS currently supports three major application programming interfaces or APIs. They are:

1. RTEMS Classic API
2. POSIX
3. High Performance API

RTEMS Classic API

The Classic API is the original API in RTEMS and is based on the VITA RTEID 2.1 standard. It is a classic, real-time operating system programming interface and provides:

1. Tasks
2. Semaphores
3. Message Queues
4. Events
5. Barriers
6. Interrupts
7. Time
8. Timers
9. Rate Monotonic Periods
10. Fixed Allocation Memory Pools
11. Variable Allocation Memory Pools

The RTEMS Classic API has a low overhead and is often used in small resource-limited targets needing a small footprint.

POSIX

RTEMS POSIX support is divided into three parts based upon where they are implemented. They may be implemented by the:

1. C Library,
2. RTEMS, or
3. TCP/IP stack.

RTEMS uses the same Newlib C library as Cygwin to provide the core C Library and most of the non-thread related POSIX capabilities. This provides a robust implementation of core POSIX and C Library services including math, stdio, and strings, and even wide character support. Much of the Newlib source code originated from BSD operating systems, but has been ported to many target architectures.

Importantly, RTEMS relies upon Newlib header files defined by the C and POSIX standards. Historically, Newlib did not provide a complete set of C and POSIX header files. It only provided those where it implemented some methods. However, there has recently been a push by the RTEMS developers to grow this set and to ensure they are compatible with FreeBSD-kernel and user-space source code.

RTEMS provides the implementation of all concurrency and synchronization capabilities including threads, mutexes, condition variables, semaphores, and message queues. It also implements system calls such as `open(2)`, `read(2)`, etc. In addition, RTEMS provides other POSIX capabilities such as termios, clocks, and timers.

RTEMS relies upon the FreeBSD TCP/IP stack to provide the networking APIs required by the POSIX standard. The older IPV4-only stack does not provide all of the capabilities required by POSIX. However, the new TCP/IP stack provided by the LibBSD project deserves credit for providing complete support.

As a single-process, multithreaded operating system, RTEMS is aligned with POSIX profiles PSE 51 and 52. These profiles define the services provided by single-process, multithreaded POSIX implementations. PSE 52 includes filesystem support, while PSE 51 does not. RTEMS applications may optionally disable all filesystem support. Thus, RTEMS can be user configured to align with either profile. FreeBSD is a multiprocess, multiuser POSIX implementation and aligned with POSIX profile PSE 54.

The Open Group Future Airborne Capability Environment (FACE™) Consortium has defined four new POSIX profiles to address the requirements of the avionics software community. These profiles reflect existing real-time operating systems and applications that have achieved certifications in industries such as avionics and medical devices. These profiles reduce the approximately 1,300 methods in the POSIX standard to meet the certification and application requirements found in existing avionics applications:

- The Security Profile is small with only 163 methods required. It is designed for multithreaded, single-process applications such as an information gateway device.
- The Safety Base Profile is larger with 246 methods required. It is designed for multithreaded, single-process applications.
- The Safety Extended Profile is larger with 335 methods required. It is designed for multithreaded, multiprocess applications.
- The General Purpose Profile is larger with 812 methods required. It is designed for multithreaded, multiprocess applications which may not have any certification requirements or less rigorous ones.

Being a single-process, multithreaded operating system with a long history of standards support, RTEMS naturally aligns with the Safety Base Profile. When initially evaluated, RTEMS was missing less than 10 methods from this profile. There is currently an effort to integrate RTEMS and the ARINC 653 Deos RTOS and achieve FACE conformance for the Safety Base Profile.

Interestingly, when evaluated against the single-process FACE General Purpose profile, RTEMS does surprisingly well, supporting approximately 90% of the methods required. Proper support for required capabilities such as `fork(2)/exec(3)` and process groups are beyond the target profile for RTEMS. However, most of those missing methods are methods that do not require multiprocessing. Newlib does not support fenv.h or long, double-complex math. These account for most of the missing methods that RTEMS could support.

High-Performance API

The close coupled High-Performance API is new and not considered a general API for use in applications. It is used in places where speed and compatibility can be traded off. The FreeBSD port is an example as well as back ends for C11/C++11 threads and OpenMP. C11/C++11 threads currently have the lowest space and time overhead in RTEMS.

Services

Services provide functionality to help developers create useful applications. The support can range from implementation of specific protocols such as

SNMP, additional languages such as Lua or Python, or important services like NTP.

Board Support Packages

A Board Support Package or BSP implementation is the code and support needed to implement RTEMS on a specific piece of hardware. RTEMS contains over 170 BSPs on 17 architectures. The BSP manages the entry from the boot loader, setting up of memory, caches, console, and it contains a timer driver for the system tick. If the processor has more than one core, then it also manages the starting of the extra cores.

While most FreeBSD users will never need to delve into the details of the device drivers for their hardware, RTEMS users often use custom hardware and thus are responsible for developing the drivers for their own boards.

FreeBSD in RTEMS

FreeBSD is an important part of RTEMS and its history. BSD code in RTEMS can be traced back to code and files being added to the Newlib C library. Since then RTEMS has taken FreeBSD directly into its source base in a number of areas. As well as the C Library, RTEMS shell commands such as `rm`, `cp`, and even `dd` are among the growing number of consumers of this code.

Networking Stack

In the late 1990s, Eric Norum, working at the Canadian Light Source in Saskatoon, started to look for a networking stack for RTEMS. Eric is a member of the EPICS project, which requires a networking stack. His first efforts included porting a stack called KA9Q, which was not that successful from both a performance and licensing viewpoint. At that point in RTEMS's history, applications were always statically linked with the OS. Also, embedded real-time applications are shipped integrated with hardware devices. There is generally no desire to redistribute any source code for the application or any supporting software. These deployment characteristics have always led the RTEMS Project to carefully evaluate the license for software incorporated directly or provided as third-party add-ons. For KA9Q, in addition to performance issues, there was never clarity on precisely what the license of the KA9Q stack actually was. This resulted in the KA9Q effort being abandoned.

Eric Norum started to look at porting the Linux network stack. However, that effort did not last because of the license, so it was suggested he look at the FreeBSD stack. Over the course of six months Eric ported the FreeBSD networking stack to RTEMS. This stack is still present in RTEMS and is essentially unchanged with specific bug fixes made as required. This network stack has been very successful and has given RTEMS all the networking sophistication available from a standard FreeBSD installation.

FreeBSD's stack is fully featured, and when ported to an RTOS users can create applications which have a system-level robustness. There are very few system configurations that cannot be done in RTEMS—from a single flat network endpoint to routing, DHCP, radio VoIP, and IP over GRE tunnels over SDH management channels.

This initial port, or what is now called the legacy port, of the FreeBSD networking code used a three tasks model. A networking task or thread runs within the networking stack and lets the stack handle things like ICMP packets and TCP retransmission. Each interface has a receive and transmit task. The receive task receives data from the MAC delivering it to the stack and the transmit task empties the output queue sending the `mbufs` to the MAC. All networking tasks have the same priority and need to hold a single networking semaphore when running. This design means there is limited networking concurrency, and throughput with more than one interface is limited, as the semaphore serializes all processing. However, the implementation is safe and reliable in a re-entrant threaded system and in practice it works very well.

Maintenance

In the years following the completion of this port, some issues began to be observed. The first and most obvious was the availability of drivers. The custom driver support required custom drivers to be written for RTEMS. Code could be borrowed from FreeBSD; however, drivers often needed to be written and tested as if new. As time moved on and FreeBSD evolved, it was frustrating and confusing to RTEMS users that a port of the FreeBSD stack did not support drivers from newer FreeBSD versions.

When the code was ported to RTEMS, it was copied into a simplified directory structure. The FreeBSD directory tree is large and wide and there seemed no point in creating a wide, sparse tree to host a small collection of files. This made it difficult to easily compare files with the FreeBSD originals because it was never really clear which files in the RTEMS source should be compared against which files in the FreeBSD source. Code was changed to get it to build and this was done without any clear indication of what was original and what was changed. As time went on and FreeBSD improved its stack, for example,

adding IPv6, it was practically impossible to move from the snapshot in the RTEMS source tree. Specific bug fixes were brought in by hand for localized fixes, but this was a time-consuming process and narrowly focused. This resulted in a maintenance problem that grew increasingly worse as the code aged.

Till Straumann, working at Stanford's SLAC National Accelerator Laboratory, built an additional library called `libbsdports` that allowed drivers from FreeBSD to be used with minimal changes. This was remarkably successful and it raised the idea of being able to use FreeBSD code with minimal changes. His work was limited to networking drivers and on a specific range of architectures, but was the first public example of this being achievable.

There were other FreeBSD activities happening around this time. Chris Johns took the USB stack and ran it on RTEMS on a NIOS-II. However, these were isolated, specific, and of no long-term value to the RTEMS Project. It also highlighted the issue of having separate ports of specific pieces of the FreeBSD kernel. How are the separate pieces brought together in a single static executable? RTEMS needed a single, unified port of the FreeBSD kernel code for all the different parts of interest. Consequently, a plan was developed to address the use of FreeBSD source in RTEMS. The first decision was to take a step back and not fragment the effort. The various parts of the FreeBSD code base that would be used by RTEMS needed to be together and maintained as a single entity. This project was named RTEMS LibBSD.

RTEMS LibBSD

The most recent FreeBSD porting project in RTEMS is called RTEMS LibBSD or simply LibBSD. The project is hosted in a separate Git repository within the RTEMS Project's Git server. The repository is at https://git.rtems.org/rtems-libbsd.git. It is a combined effort led by Joel Sherrill and Sebastian Huber.

The project creates a single port of FreeBSD for RTEMS and provides a range of features present in FreeBSD useful to RTEMS such as networking, USB, SATA, and MMC devices.

As the effort progressed, a broad set of rules was developed to guide developers working on RTEMS LibBSD. The rules slowly took shape as the team found what worked and what did not work. The rules can be summarized as:

1. The directory structure of the RTEMS LibBSD code must match the source tree in FreeBSD.
2. All changes in the RTEMS version of the code must be bounded by a standard conditional defined syntax. This allows removal of the RTEMS changes and comparison of the source with the original FreeBSD code using Python scripts.
3. Do not edit the FreeBSD code including any white-space changes. Make all edits in preprocessor conditionals.

The ability to use original FreeBSD source transparently is central to the work done in LibBSD and the term "source transparency" has been adopted to describe the approach. Anyone embedding FreeBSD code with their own system and machine headers wants to be able to take a subset of the FreeBSD source files and build them without any changes. Currently this is not possible. When viewing the FreeBSD source code from the RTEMS Project's point of view, any transparent source has no changes, and as changes are made, the original FreeBSD becomes less visible.

The major items to resolve when embedding FreeBSD kernel code are:

1. Header files and required declarations. The system and machine header files for an RTEMS target do not match the header files used by the FreeBSD kernel code. The improved standardization of headers has helped on both sides, but there is a range of kernel types and defines that need to be added.
2. The use of standards-based, userland function names with differing signatures in the kernel, for example, malloc. RTEMS is a single address space, statically-linked executable, and these name clashes need to be managed and often are—with horrible hacks.
3. Supporting the SYSINIT initialization used in FreeBSD. This requires linker support to get the correct section management in place. The way this is done in FreeBSD is so good that something similar was adopted in the RTEMS kernel and with impressive results. RTEMS has adopted the linker mechanism to initialization used by FreeBSD with SYSINIT. Previously RTEMS required users to manage in their build system the parts of RTEMS they needed linked in, and for those parts they did not want, they needed to link dummy versions. With the linker set initialization this is all automatic, including the order in which the initialization happens, making RTEMS simpler to use. Although there had always been a focus on small executable size, the size of the RTEMS minimum reference application decreased in size thanks to the change to SYSINIT style initialization.
4. SMP support requires that some parts of the FreeBSD port be managed in the context of RTEMS SMP support.

CONTINUES NEXT PAGE

5. FreeBSD userland code in a single, statically-linked executable requires some interesting hacks to avoid global symbol clashes and to make initialized variables work. This is especially obvious when porting shell commands into a single address space. Each command has its own `main()`, and calling `exit()` does not exit the command invocation—it exits the entire RTEMS application.

Source Code Management

The source tree consists of four major directories. They are:

1. `freebsd` – RTEMS's FreeBSD source code
2. `rtemsbsd` – RTEMS's FreeBSD support source code
3. `testsuite` – Tests for RTEMS LibBSD
4. `rtems_waf` – RTEMS's waf support for building against RTEMS BSPs.

When using LibBSD, the Git sub-module for `rtems_waf` must be initialized to bring its support files into the cloned repository. The module helps configure and build LibBSD for a BSP. A RTEMS toolchain and suitable board support package (BSP) also needs to have been built and installed.

Developers working on LibBSD will also need to clone the FreeBSD source tree from the snapshot point. This creates an extra directory called "freebsd-org."

The RTEMS LibBSD project has around 850 build targets. To manage this number of files and the complex set of compile time defines, architecture specific files, and header files, the definitions are isolated into a single file that is independent of the build system. The initial development generated a `makefile` and recently this was changed to generate a `waf` script (https://www.waf.io). The `waf` build script integrates with the "`rtems_waf`" support, which makes it easier to extract and use the various machine specific compile flags a BSP has. As RTEMS has a single address space, it is critical that the operating system, user code, and any libraries like LibBSD be built using the same ABI.

The LibBSD build definition is a single Python file called `libbsp.py`. The file contains a number of module definitions and these modules are passed to the source management tool to manage moving source in and out of the original FreeBSD source tree to the RTEMS FreeBSD source tree. The module data is also passed to the build script generator to create the `waf` build script.

There is support for a number of different types of modules. Modules can be for RTEMS headers and source, FreeBSD kernel space headers and source, or FreeBSD user space header and source. The module definitions can be given specific compiler flags, or they can be specialized for a specific architecture. A lot of the source is generic to all architectures; however, some files are specific to some architectures, for example, an architecture-specific IP checksum routine.

To add new source files to LibBSD, first initialize the FreeBSD Git sub-module, which populates the freebsd-org source tree with the specific version LibBSD is currently based upon. Run the "freebsd-to-rtems.py" script in reverse to move the modified source in RTEMS FreeBSD to the original FreeBSD source tree. Edit the source definition to add the new files and run the same script in the forward direction. The source will be copied across to the LibBSD tree and the build script will be updated. LibBSD can now be built and the source edited so it can run.

The user space symbol clashes are managed by maintaining a header file that redefines the symbols to a different name space. This is not ideal, but it can be made to work. The clashing symbols, however, raises a common issue, the need to include some header files before any FreeBSD code appears. Having to change the standard code to include headers accounts for a reasonable percentage of the changes made to the original source code. An empty header in a normal FreeBSD kernel tree would help this type of porting exercise because it could include an RTEMS-specific version that includes the symbol refines. It is a simple change in the kernel that makes a larger difference to the RTEMS project.

FreeBSD Core APIs and RTEMS Mappings

The FreeBSD kernel support is implemented in terms of RTEMS services. The following describes those mappings:

1. Shared/exclusive locks `SX(9)` map to binary semaphores. This neglects the ability to allow shared access.
2. Mutual exclusion `MUTEX(9)` maps to binary semaphores. Non-recursive mutexes are not supported this way.
3. Reader/writer locks `RWLOCK(9)` map to binary semaphores. This neglects the ability to allow multiple reader access.
4. The sleep queues `SLEEPQUEUE(9)` use the FreeBSD implementation with adjustments for RTEMS.
5. Condition variables `CONDVAR(9)` use the FreeBSD implementation (no signals support).

CONTINUES NEXT PAGE

6. Timer functions `CALLOUT(9)` mainly use the FreeBSD implementation. The wheel driver is a RTEMS's timer server routine.
7. Tasks `KTHREAD(9), KPROC(9)` map to RTEMS tasks.
8. Device management `DEVCLASS(9)`, `DEVICE(9)`, `DRIVER(9)`, `MAKE_DEV(9)` uses the FreeBSD implementation.
9. Bus and DMA access `BUS_SPACE(9)`, `BUS_DMA(9)` maps to Board Support Package implementations. A default implementation for memory-mapped, linear access is provided and the current RTEMS heap implementation supports all the properties demanded by `bus_dma`.
10. The Universal Memory Allocator `UMA(9)` is supported using a simple page allocator as back-end allocator.

From a maintenance perspective, the RTEMS developers hope that these FreeBSD kernel APIs remain stable. This reduces the risk that updates to FreeBSD require significant work to the RTEMS implementation of the core kernel APIs.

User Space

An important part of using FreeBSD is the user-space support. The commands such as `sysctl`, `ifconfig`, and `netstat` provide an important user interface. Being able to support these types of commands not only makes using the FreeBSD software possible, it allows RTEMS to tap into the wealth of existing FreeBSD documentation. This is an important, but often overlooked, source of reuse.

Porting separate user space programs into a single executable is an involved and delicate operation. As RTEMS has only a single address space available, neither `fork(2)` nor `exec(3)` are available to set up a new process context. Many common assumptions that C code makes cannot be relied on. RTEMS has to emulate a subset of both `fork(2)` and `exec(3)` semantics. This is accomplished with a combination of build-time, link-time, and run-time techniques. LibBSD provides some abstractions in the form of wrapper functions to help.

When bringing this type of code into RTEMS, it is critical that there are no globals. A range of prepossessing tricks can be used and none of them are nice. Any global data needs to initialized for each run of the command. The use of `getopts` needs to be replaced with a nonstandard re-entrant version provided by Newlib. Finally, `main()` is replaced with a function name such as `main_dd()` for the command. A few important other functions like `exit()` or `perror()` are also replaced with calls to the LibBSD command support code.

The generic support for the commands in LibBSD only allows a single command to run at once. While this is not ideal, multi-user access to RTEMS is normally not an issue. The generic support wraps the call to the specific `main()` in a `setjmp()` call. An `exit()` call is implemented by a `longjmp()` call. Since RTEMS is a single-process, multithreaded environment, executing the native `exit(3)` in an application shuts down the entire application.

The resulting code is not pretty in places; however, the results are rather impressive. There is a working network stack with related FreeBSD commands including `ifconfig`, `netstat`, `ping`, and `tcpdump`, which is an interesting command to run on a real-time operating system. Full Gigabit Ethernet performance has been sustained for both transmit and receive on ARM, PowerPC, and x86 systems.

Initialization

As the code in LibBSD becomes more stable and supports more BSPs, users are beginning to migrate to it and RTEMS developers are working on the boot-time initialization. To solve this problem, the decision was made to adopt the standard methods used by FreeBSD.

Recently Chris Johns added code to support `rc.conf(5)` into the source tree. This is a C implementation, as RTEMS does not have a POSIX sh. The benefit of supporting `rc.conf(5)` to initialize the code is the large amount of documentation and solutions available on the Internet. This is a large saving in effort for the RTEMS project in terms of documentation.

Chris Johns also ported `sysctl(8)` as a command and is in the process of adding support to read `sysctl.conf(5)`. As this support matures, RTEMS-specific code to manage various memory size configuration options will be removed. This will both improve portability and lessen the differences in using and configuring LibBSD and FreeBSD.

The Future with FreeBSD

How good is LibBSD currently? The answer to this question depends on the perspective taken.

Any FreeBSD user who needs to use an RTOS will be pleased by the LibBSD efforts. They have a full-featured, high-performance, stable networking stack with IPv6, packet filtering, virtual LAN support, and more with world class performance. Users have access to a wide range of

drivers, and device drivers tend to port quickly with few changes.

The RTEMS developers are far more critical. The ideal is being able to use the FreeBSD code with no changes. While this may never happen, it is critical to ensure that the team strives for it because it lowers the maintenance burden. LibBSD is currently stuck on a FreeBSD 9.x version because updating will require more effort than is available. This puts LibBSD two major FreeBSD releases behind (e.g., FreeBSD 10 and 11). This is due to a combination of issues, some changes in FreeBSD and some code that is not clearly tagged.

There are currently 1,295 source files in LibBSD, and, of those, 797, or 61% of the files, have no changes. Of the 498 files with changes, there is an average opacity level of 1.6%. The level of opacity is a simplistic calculation made by taking the number of inserts and deletes as a percentage of the total lines of code including the inserts and deletes. There are less than 50 files with an opacity level greater than 10%. It is fair to say this is a reasonably successful result.

What remains to be done? (list follows)

- Upgrade the current code to better track FreeBSD. It will always be necessary to spend some effort on this. However, RTEMS developers hope that by making a strong case to FreeBSD kernel maintainers for some simple and helpful changes, the LibBSD maintenance burden will be manageable.
- The number of architectures supported by LibBSD needs to be increased. LibBSD currently is known to work on three architectures, while RTEMS itself supports 18 architectures. RTEMS is a good base reference for building on architectures not normally accessible.
- Although not critical from a functional viewpoint, compile-time warnings in LibBSD are currently ignored. The RTEMS build environment is significantly different from the normal FreeBSD kernel environment. However, it would be nice to see the warnings addressed. As always, many warnings are harmless, but some could be indicative of real bugs.

The FreeBSD code base is important, and there are a number of users of the source code reusing it in ways the original authors never imagined. The RTEMS Project openly and at every possible opportunity acknowledges the code RTEMS has taken from FreeBSD. It is an incredible resource and the RTEMS Project is thankful it is available for use. ●

References

VMEBus Industry Trade Association (VITA) Real-Time Executive Interface Definition (RTEID) 2.1 standard. The RTEID 2.1 is archived at ftp://ftp.rtems.org/pub/rtems/people/joel/RTEID-ORKID/RTEID-2.1/.

Newlib C Library. https://sourceware.org/newlib.

Cygwin. https://www.cygwin.com.

G. Gilliand, J. Sherrill. "A Unique Approach to FACE conformance." U.S. Army Aviation FACE Technical Interchange Meeting. http://face.intrepidinc.com/wp-content/uploads/2016/01/DDC-I-OAR-A-Unique-Approach-to-FACE-Conformance.pdf. (Feb. 2016)

"Proceedings of Technology Showcase Held in Huntsville, Alabama on 7–9 August 1990," includes a presentation by a representative of the Army Missile Research Development and Engineering Center at Redstone Arsenal, Alabama, on RTEMS. http://www.dtic.mil/dtic/tr/fulltext/u2/a247043.pdf. (Aug. 1990)

KA9Q. http://www.ka9q.net/code/ka9qnos/.

A. Subbarao. "POSIX—25 Years of Open Standard APIs." http://www.rtcmagazine.com/articles/view/103514.

FACE Consortium products including the Technical Standard, Conformance Test Suite, and other supporting artifacts. https://www.opengroup.org/face.

CHRIS JOHNS is a RTEMS developer and real-time software engineer with an interest in low-level tools and debuggers. He lives in Sydney, Australia, with his family and two dogs. chrisj@rtems.org

JOEL SHERRILL is Director of Research and Development at OAR Corporation in Huntsville, Alabama, and a member of the original RTEMS team. joel@rtems.org

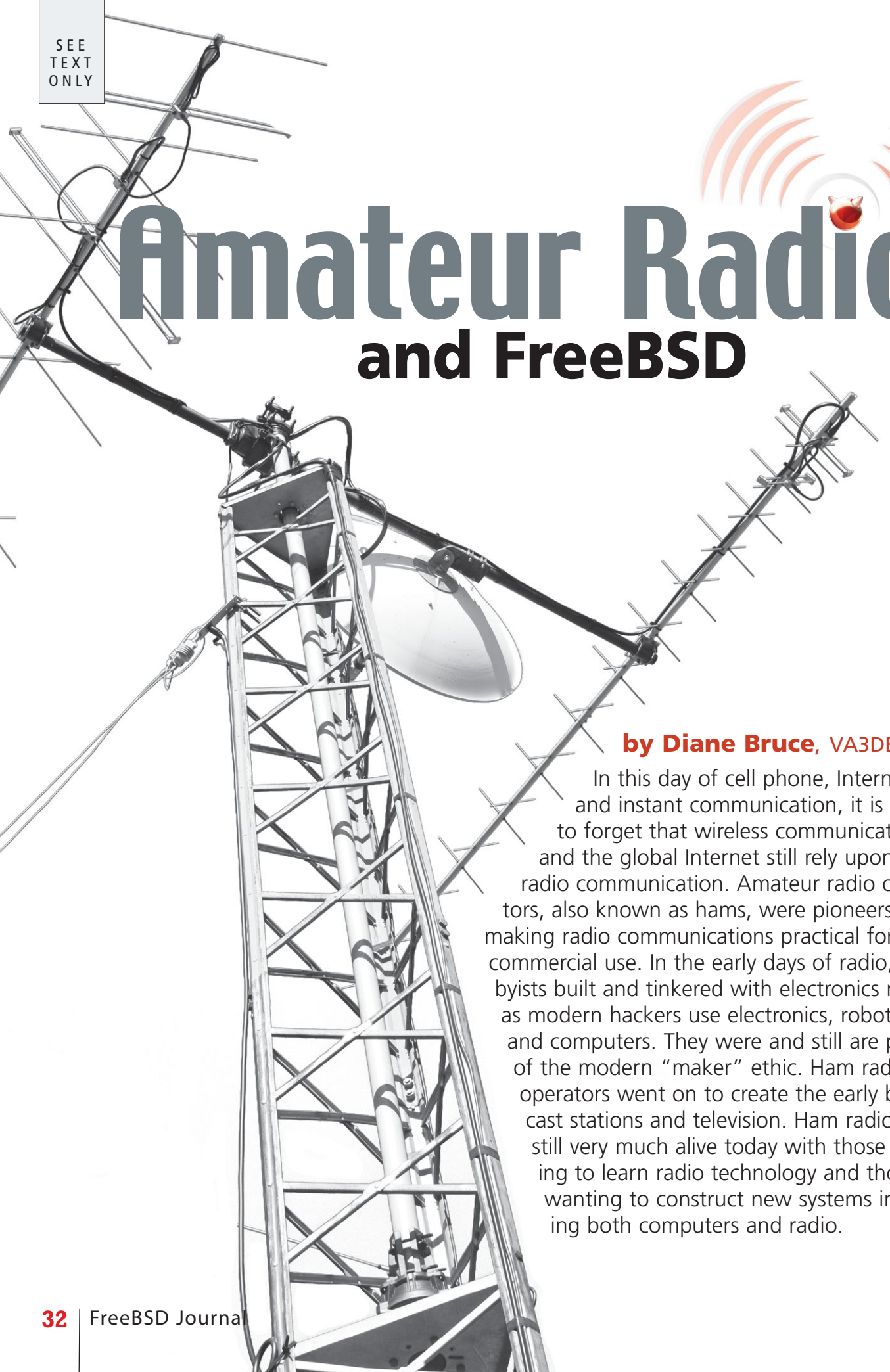
SEBASTIAN HUBER is a RTEMS developer currently focused on the SMP support. He lives in Munich, Germany. sebh@rtems.org

BEN GRAS is a systems security researcher at VU University in Amsterdam, Netherlands, and an RTEMS contributor. He lives in Amsterdam. beng@rtems.org

GEDARE BLOOM is an assistant professor of computer science at Howard University in Washington, D.C., and a maintainer of RTEMS. gedare@rtems.org

SEE TEXT ONLY

Amateur Radio and FreeBSD



by Diane Bruce, VA3DB

In this day of cell phone, Internet, and instant communication, it is easy to forget that wireless communication and the global Internet still rely upon radio communication. Amateur radio operators, also known as hams, were pioneers in making radio communications practical for commercial use. In the early days of radio, hobbyists built and tinkered with electronics much as modern hackers use electronics, robotics, and computers. They were and still are part of the modern "maker" ethic. Ham radio operators went on to create the early broadcast stations and television. Ham radio is still very much alive today with those wanting to learn radio technology and those wanting to construct new systems involving both computers and radio.

Amateur radio is a federally licensed hobby. As the radio spectrum is shared among many services, and hams are involved in experimentation, a basic knowledge of radio technology and regulations is important to prevent interference with other services. Amateur radio is one of the few radio services that are allowed to build their own radios, new protocols and systems.

Licensing requirements are easily met by anyone with a technical background, and no, Morse code is not needed.

Amateurs have worked on cell phone development and satellite systems, as well as helping build the modern Internet. Scientists, computer programmers, and electronics engineers all have a keen interest in amateur radio.

But why use computers with ham radio in the first place? The technology of computers and radio itself has changed radically over the past few years. The modern radio ham uses computers to handle satellite prediction, digitally encoded voice, logging, digital modes, software-defined radio, and a myriad of other tasks.

This has resulted in many applications written for amateur radio use, many written to run on the Linux operating system. Those of us (https://wiki.freebsd.org/Hamradio On FreeBSD) who work on ham radio ports for FreeBSD would love to change this attitude and bring BSD into the office as well. Fortunately, many of the applications written generically or specifically for Linux are easily transferred to FreeBSD.

`fldigi` **running on FreeBSD desktop.**

Ham radio operators were early adopters of personal computers for use in sending and receiving radio teletype signals. Radio teletype (RTTY) for the amateur in the early days utilized surplus, obsolete teletype machines such as the model 15 with an external modem.

You can imagine how these clunky machines made RTTY impractical for many amateurs! These machines used a predecessor to modern 8-bit ASCII code often called Baudot (Murray code for the pedantic), which was a 5-bit (5-level) plus start/stop bit code. These machines were then coupled to a modem made for radio use that did extensive filtering of the audio signal to reduce interference from other signals. RTTY was a natural for the early home personal computer and is easily one of the earliest digital modes. It was very easy to generate and decode 5-level code using an early 8-bit computer such as an Apple II, but still using the external modem. Computer power has reached the point where it is trivial to use signal processing instead of an external modem to decode off-the-air, radio teletype directly, and display the decoded text. A typical program used for this with FreeBSD is fldigi.

fldigi itself is the modern Swiss army knife of many off-the-air short wave signals such as RTTY and Hellschreiber and more modern protocols such as PSK31.

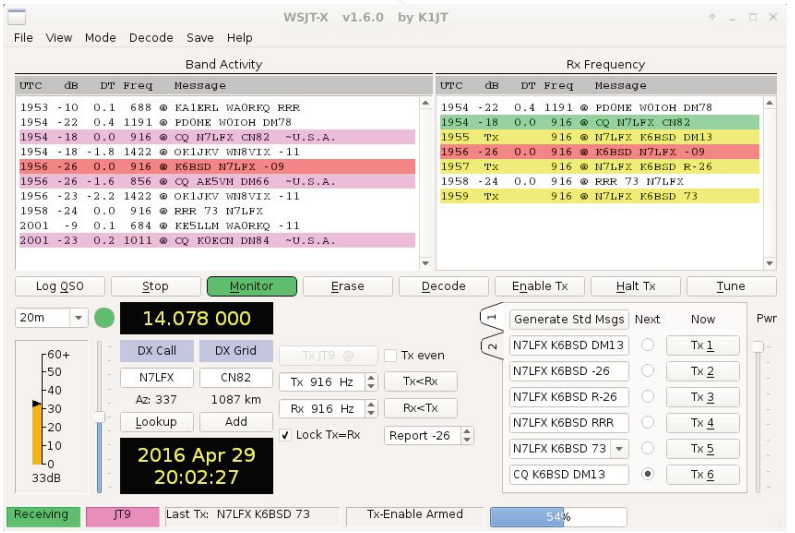
wikipedia photo

Model 15 teletype.

Old 1950s ham radio operator's station.
http://www.gladlylearn.com/Amateur Radio/RSGB/RSGB_39_Late1950s.JPG

wikipedia photo

Hellschreiber machine from WWII. Tones were used to draw characters upon a moving drum.
For more information, see wikipedia https://en.wikipedia.org/wiki/Hellschreiber.

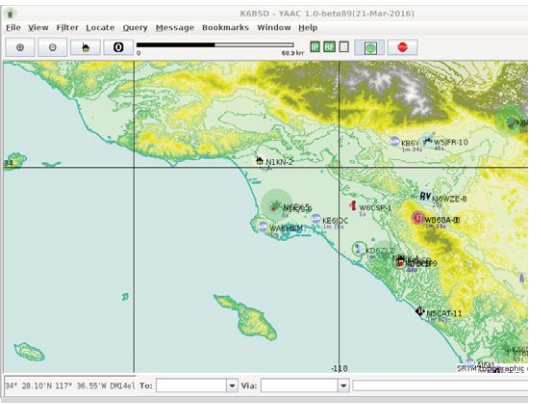


WSJTX used on 20 meters.

Typical antenna setup needed without WSJT.

With the increased power of modern computers, weak signal detection using modern digital signal processing has improved immensely. Joe Taylor, a Nobel Prize-winning physicist who is also known by his ham radio call sign K1JT, wanted a way to send signals to the moon and back (Earth-Moon-Earth or simply EME or moon-bounce) to send signals around the globe via the moon. Using his expertise in radio astronomy, he used modern advanced signal processing to develop Weak Signal JT (WSJT), using a new mode JT65. Early ham radio EME operations required very expensive and large antenna arrays with high-powered amplifiers. WSJT brings EME to amateurs using a more modest station that is far less expensive to set up.

Ham radio operators all over the world now use WSJT, WSJTX, and its offspring, WSPR, in daily use and to communicate with very low power around the globe and not just via the moon, but by traditional



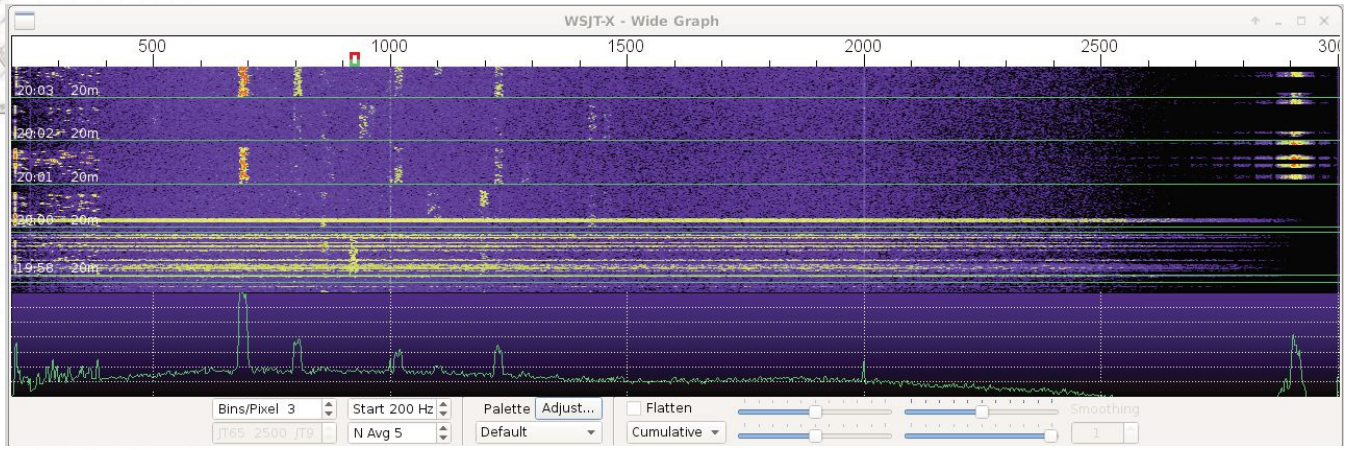
Map of stations and locations as seen by YAAC (west coast of USA).

short wave using the ionosphere. (See http://physics.princeton.edu/pulsar/K1JT/)

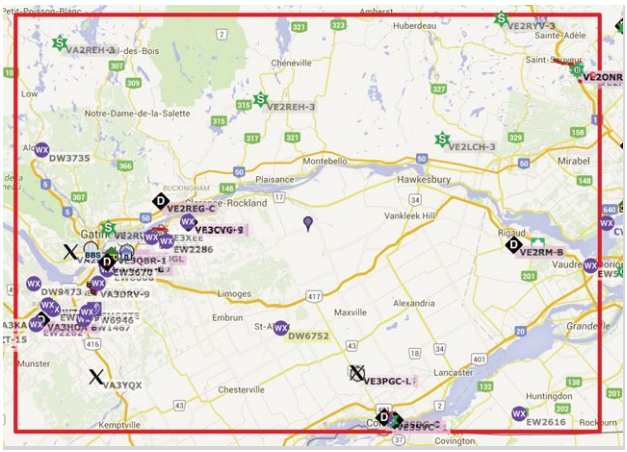
A recent trans-Atlantic 2m attempt was found to have succeeded simply due to the signals bouncing from the International Space Station, which just happened to be in the right place at the right time! (July 14, 2014 see http://www.brendanquest.org/)

PSK31 is a low-bandwidth mode that is also very popular with modern hams. Its ability to be heard below the noise floor makes it very popular for low-power operators. Again fldigi is the program of choice here for most hams.

Amateur radio operators were instrumental in early work with packet radio, which has found its way into encrypted digital systems for police and other emergency services. Store and forward networks using a modified X.25 protocol, AX.25, are still in use all over the world. This forms the backbone of the tracking system known as Amateur Positioning



WSJTX in so-called waterfall display showing region of frequencies vs. time.



View of stations in Ottawa Ontario, Canada using APRS as seen by http://aprs.fi.

Radio System (APRS). This system is well supported from FreeBSD using `Xastir` and `YAAC`.

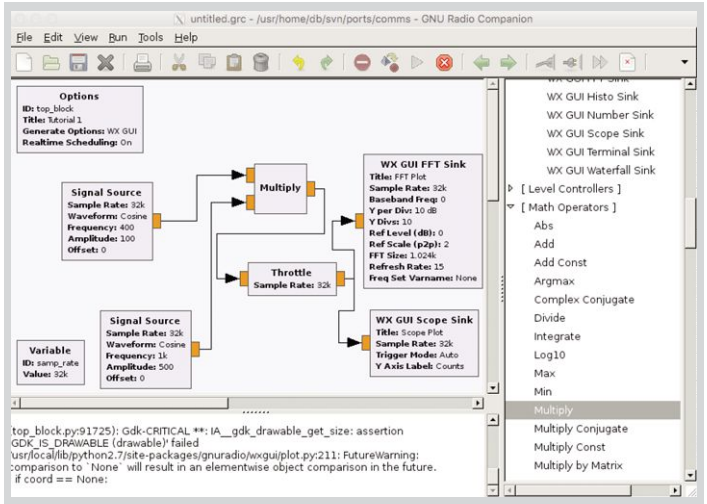
Stations use GPS receivers to broadcast their current positions via APRS over AX.25. This system is invaluable for volunteers using amateur radio who help the professionals who do search and rescue—Civilian Air Patrol (CAP), for example. These signals are also relayed on the Internet to allow worldwide tracking of stations; e.g., http://aprs.fi/#!addr=FN25 will show my neck of the woods in the Ottawa area.

Software-defined radio is one of the hottest new techniques being experimented with by radio amateurs. By using fast A/D converters, radio signals can be sampled directly from the air, converted to quadrature (two signals with a 90 degree phase shift between them) digital samples, commonly referred to as I/Q signals to be decoded using computers. For the radio, amateur signals can also be generated using D/A converters and transmitted.

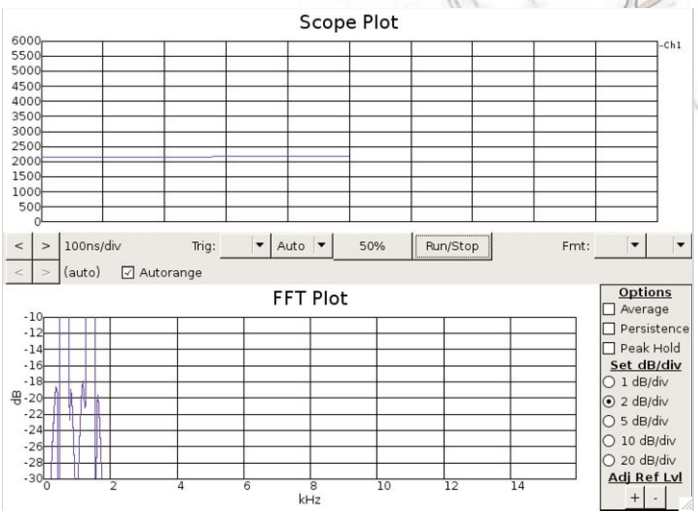
When Adrian Chadd (ham radio call sign KK6VQK, adrian@FreeBSD.org) wanted to look at the layout of frequencies used by WiFi transmission, he needed a software-defined radio (SDR) package that was well supported. That meant Adrian had to help port software to support the Ettus systems USRP to FreeBSD if he wanted to use it on his BSD systems with `gnuradio`. `gnuradio` is a software framework of components that can be linked together using a graphical interface to put together SDR radio systems.

High end RF A/D D/A systems can handle many Mhz of frequencies at once, much as you would have to do with radio astronomy or in WiFi signal analysis. However, much SDR can be done using the standard audio card that comes with any modern computer system or using a DVB-T TV tuner USB dongle based on the RTL2832U chipset. The sound card is limited to sampling baseband audio signals; hence, any RF signals must be down converted to base band before they can be decoded. The so-called "SoftRock" is a low cost RF converter that can be used with programs such as QUISK to decode traditional short wave modes including single sideband (SSB), FM, and AM.

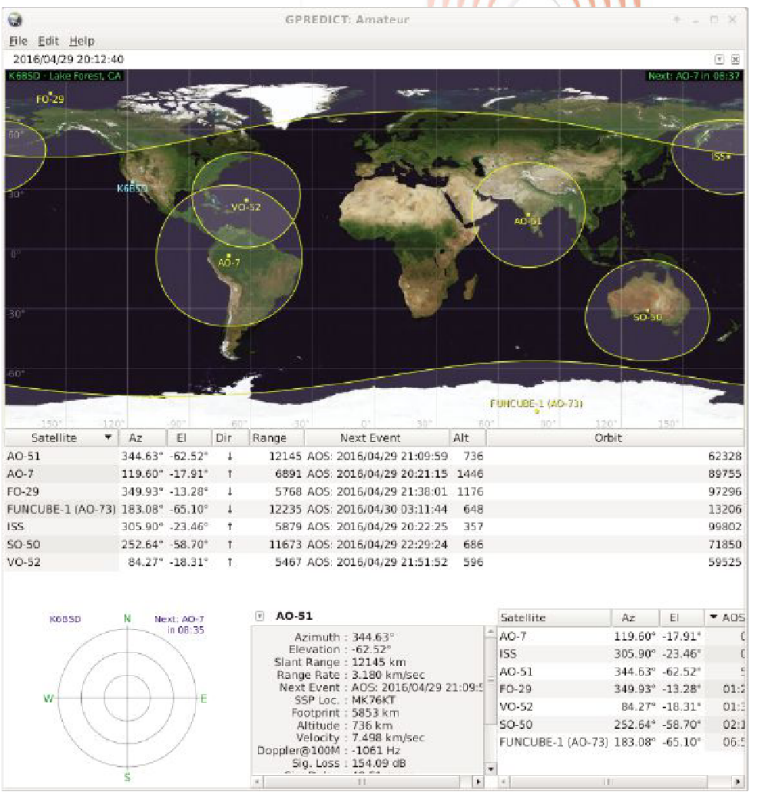
A TV tuner USB dongle can be used to directly sample RF signals well up into the UHF radio spectrum, and as such can be used to monitor ham radio frequencies in this spectrum or even just tune in a broadcast FM station; e.g., the `rtl-sdr` port can be used in conjunction with `gnuradio`. Many ham radio enthusiasts have



`gnuradio-companion` **graphical interface being used to set up a simple DSP tutorial.**



`gnuradio` **display of scope and fft plot of sample generator.**



`gpredict` satellite prediction program shows coverage areas of satellites being tracked.

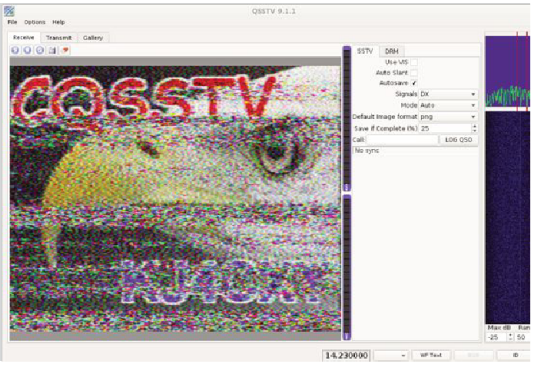
Early amateur radio built satellite AO-7.

built-up converters to GNU Radioas used by gnuradio.org that convert low, short-wave received frequencies up into the region that a TV tuner dongle can receive.

Ham radio operators have designed and built their own satellites. AO-7, first launched in 1974, is still going, albeit in a severely degraded mode as the batteries have now died.

Building a satellite takes knowledge in diverse fields, such as power engineering, battery technology, radio, and embedded computer systems. These are the sorts of people who work for NASA. For the average amateur radio operator, knowing where to aim the antenna and at what times for each satellite takes tracking software. Such programs as predict and gpredict are commonly used here.

The International Space Station has licensed amateur radio operators on board and in a low-earth orbit. It is a very easy station to hear when it is in automated mode and to talk to when they are active. As can be expected, the astronauts on board don't have a lot of time to spare for talking directly to ham radio operators on the ground, but often will make special arrangements to talk to schools on the ground. You can see this satellite listed in the screenshot of `gpredict`.



`QSSTV` program used to decode slow scan TV.

Amateur radio television is a mode that is in use by some hams, though most of the activity is with a low bandwidth form of television called slow scan TV (SSTV) SSTV in the early days required surplus radar tubes with a long persistence (P7) phosphor that was very harsh on the eyes, but would allow the picture to be painted before it faded. Nowadays, this is all done using computer decoding and allows us full color pictures. The ISS has been known to broadcast SSTV signals to the earth for ground stations to view as well.

Repeaters are used to extend the distance of mobile stations by putting a receiver on a hill or tall building along with a transmitter that relays the signals from the mobile stations. It is a trivial matter to link local city repeaters across the world via the Internet. This can be done using `thebridge` or `svxlink`.

Amateur radio can be as technical as you want it or just a relaxing hobby. There are so many diverse interests in the hobby with so many different aspects that I can only cover a small part of it. The era of inexpensive computing has, and is, making amateur radio much more interesting. Perhaps you yourself now have an interest. For further reading I would suggest starting with http://www.arrl.org in the U.S. and http://www.rac.ca in Canada.

DIANE BRUCE has done computer programming in one form or another for over 40 years, and has 35+ years' experience in embedded/real-time. Diane currently contributes to the FreeBSD Project, mostly in ports. Her hobbies include amateur radio, writing, and music (amateur musician). She was first licensed as a radio amateur in 1968. db@Freebsd.org

Thank you!

The FreesBSD Foundation would like to acknowledge the following companies for their continued support of the Project. Because of generous donations such as these we are able to continue moving the Project forward.

FreeBSD FOUNDATION

Are you a fan of FreeBSD? Help us give back to the Project and donate today!
freebsdfoundation.org/donate/

Iridium

NetApp

Gold

NETFLIX

Silver

accelerationsystems

vmware Tarsnap

Please check out the full list of generous community investors at freebsdfoundation.org/donors

SEE TEXT ONLY

svn UPDATE

by Steven Kreuzer

At the time of this writing, the code slush for 11.0-RELEASE is in effect, and by the time you are reading this, head will be frozen in preparation for a September 2 target release date. Even as we rapidly approach that date, we are still seeing heavy development, and new and exciting features being added daily. Below is a small sample of some improvements made over the past two months, but I encourage you to take a stroll through svn log and take a look at the continued work on areas such as high-performance storage subsystems, low-latency networking, security and improved support for exotic architectures such as RISV-V, ARM, and PowerPC.

Native PCI Express HotPlug Support

As of r299142 (https://svnweb.freebsd.org/base?view=revision&revision=299142), it is possible to insert a new PCI hot plug adapter into an available PCI slot and make the device accessible to the operating system and applications without having to restart the machine. PCI-express HotPlug support is implemented via bits in the slot registers of the PCI-express capability of the downstream port along with an interrupt that triggers when bits in the slot status register change. This is implemented for FreeBSD by adding HotPlug support to the PCI-PCI bridge driver that attaches to the virtual PCI-PCI bridges representing downstream ports on HotPlug slots. The PCI-PCI bridge driver registers an interrupt handler to receive HotPlug events. It also uses the slot registers to determine the current HotPlug state and drive an internal HotPlug state machine. For simplicity of implementation, the PCI-PCI bridge device detaches and deletes the child PCI device when a card is removed from a slot and creates and attaches a PCI child device when a card is inserted into the slot. PCI-express HotPlug support is conditional on the PCI_HP option, which is enabled by default on arm64, x86, and powerpc.

Native Graphics Support in bhyve

When the bhyve hypervisor was first introduced in FreeBSD 10.0 the excitement in the FreeBSD community was palpable. For a very long time, the virtualization options available for FreeBSD left quite a bit to be desired, but that soon quickly changed when a BSD-licensed hypervisor became part of the base system. At first only the serial console was supported and it lacked the ability to emulate graphical consoles, but, in r300829 (https://svnweb.freebsd.org/base?view=revision&revision=300829), support for native graphics was added. This adds emulations for a raw framebuffer device, PS2 keyboard/mouse, XHCI USB controller and a USB tablet. A simple VNC server is provided for keyboard/mouse input and graphics output. With the appropriate UEFI image, FreeBSD, Windows, and Linux guests can be installed and run in graphics mode using the UEFI/GOP framebuffer.

Fault Management Daemon for ZFS

I am happy to say that the wait is finally over! Ever since the project was first announced several years ago, anyone who is familiar with the zpool command has been waiting for r300906 (https://svnweb.freebsd.org/base?view=revision&revision=300906); zfsd(8), a daemon that deals with hard drive faults in ZFS pools has been imported into head. zfsd(8) allows FreeBSD to be able to catch and handle disk events as they happen as well as automatically manage hotspares and replacements in drive slots that publish physical paths. Storage administrators managing a few terabytes to multiple petabytes should now be able to sleep a little easier at night.

bsdinstall/zfsboot GPT+BIOS+GELI Installs Now Make Use of GELIBOOT

FreeBSD introduced support for full-disk encryption using GELI over a decade ago, but you

were still required to keep the loader and kernel unencrypted so that the GEOM module could be loaded to handle decryption. As of r300436 (https://svnweb.freebsd.org/base?view=revision&revision=300436), ZFS boot environments can now be used in combination with full disk encryption without requiring the kernel and bootloader live outside of the boot environment.

Skein Hashing Algorithm

Support for Skein as a ZFS checksum algorithm was introduced in r289422 (https://svnweb.freebsd.org/base?view=revision&revision=289422), but is disconnected because FreeBSD lacked a Skein implementation. The Skein hashing algorithm was added in r300921 (https://svnweb.freebsd.org/base?view=revision&revision=30092) and has been connected to both userland (libmd, libcrypt, sbin/md5) and kernel (crypto.ko). A future commit will enable this hashing function in ZFS, as well.

Updates to head/contrib

The base FreeBSD userland is made up of quite a few utilities, some of which are developed outside the project. In the past few months, we've seen a few updates to the 3rd-party software that help create a great user experience.

- file has been upgraded to version 5.26. (r298192) (https://svnweb.freebsd.org/base?view=revision&revision=298192)
- libucl has been upgraded to version 0.8.0 (r298166) (https://svnweb.freebsd.org/base?view=revision&revision=298166)
- sqlite3 has been upgraded to version 3.12.1 (r298161) (https://svnweb.freebsd.org/base?view=revision&revision=298161)
- clang, llvm, lldb, and compiler-rt have been upgraded to version 3.8.0 (r296417) (https://svnweb.freebsd.org/base?view=revision&revision=296417)
- libc++ has been upgraded to version 3.8.0 (r300770) (https://svnweb.freebsd.org/base?view=revision&revision=300770)
- ACPICA has been upgraded to version 20160527 (r300879) (https://svnweb.freebsd.org/base?view=revision&revision=300879)
- libarchive has been upgraded to version 3.2.0 (r299529) (https://svnweb.freebsd.org/base?view=revision&revision=299529)

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.

SEE TEXT ONLY

conference REPORT™

by Tim Moore

BSDCan—EDUCATIONAL, SOCIAL, OPEN SOURCE

BSDCan 2016 can be described as a whirlwind—a deluge of information crammed into a section of the University of Ottawa for four days. Large companies like Microsoft give slick presentations about their latest and greatest innovations, while down the hall a hobbyist excitedly shares for an hour about his successes and failures in building a home network with FreeBSD. Walking the convention hallway reveals vendor displays and conference-goers eagerly discussing some facet of the vendors' products. Small knots of programmers huddle around one or more laptops, collaborating on a project or just laughing over a silly Internet video.

The Journey Begins

My journey began on a day like so many others, with an early morning trip to the airport. One advantage to living in the Middle-of-Nowhere, Tennessee, is that the local airport is always quiet and simple to navigate. After one rather uncomfortable plane ride, my coworkers and I arrived at Newark airport, where we immediately bumped into another individual on the way to BSDCan. It is easy to find another conference-bound person at the airport when in the sea of business suits there is someone wearing a Star Wars shirt and typing furiously on a laptop festooned with BSD stickers. We boarded the flight to Ottawa, and after another uncomfortable, albeit short, flight, I was standing on sweet terra firma once more.

Oh, Canada!

Ottawa airport is great. It strikes that perfect balance between being large enough to have all the amenities (bus service, customs, currency exchange, etc.) and still small enough to be simple to self-navigate. Conveniently, the bus service runs from Ottawa airport to the university, so I decided to purchase some bus tickets and enjoy the scenery on the way to the hotel. My employer booked me at a hotel just down the street from the university, but a university residency building was also available to conference attendees as an affordable alternative. Exploring Ottawa was a delight. Despite large construction projects making the city center something of a mess, it was easy enough to walk everywhere and explore the sights. I was excited to see some kind of Star Trek convention was scheduled for the weekend, but unfortunately my returning flight prevented me from attending. Still, a charming airport, a walkable city, and interesting sights made my time in Ottawa enjoyable.

Developers Summit

Wednesday, June 8, was the first day of the pre-conference Developers Summit. The Dev Summit brings together FreeBSD contributors in one place for two days to discuss issues in the FreeBSD Project and develop solutions. After registering and receiving my name tag and other goodies, everyone gathered in one of the larger classrooms for the opening session. Following a few introductory remarks, a Microsoft representative gave a presentation about FreeBSD in Microsoft Azure. At this point it was announced that a virtual machine (VM) image of FreeBSD 10.3 would be available in the Microsoft Azure Marketplace, with Microsoft providing technical support. This generated quite

a bit of buzz in the room. Next, representatives from Intel discussed the Intel Quick Assist Driver and a FreeBSD port. This presentation was more interactive, with Intel actively soliciting feedback from the developers on the technology and ways to improve or expand its features. Once this presentation concluded, the general session broke up for lunch and smaller working group sessions.

After lunch, the various developer working groups convened in smaller classrooms. These groups generally focus on specific aspects of the FreeBSD Project and ideas for improvements. I attended the documentation working group, where the primary topics were improving the documentation website and simplifying the contribution process in order to attract more interest in the documentation project. If my working group was any indication, a lively spirit of discussion and idea generation is the norm for these sessions. Once the working groups wound down for the day, we had options for dinner and continued work via even less formal sessions which wound late into the evening.

Day 2 of the Dev Summit was similar to the first. The first general session was spent compiling a list of major changes in the impending release of FreeBSD 11. While probably not a comprehensive list of all changes, I was still struck by how much individuals and companies contribute to this open-source project. After a short recess, another list of desired additions for FreeBSD 12 was compiled so as to illuminate the path forward for the contributors. Then a final list was created of all the additions the developers had brought forth and were willing to incorporate into FreeBSD. Once again, I was surprised by the general spirit of cooperation and support for the project. After lunch, we divided into smaller working groups for the rest of the day, discussing varied topics such as the efforts to teach BSD at schools or improving performance-monitoring tools in FreeBSD.

I think it is also important to note that during the Dev Summit and Conference, there are far more working groups, presentations, or tutorials being offered than any one person can attend. As someone who generally tries to experience as much as possible during a conference like this, I am definitely planning to map out my activities much more thoroughly for future conferences.

The Conference Begins

The general structure of the conference was to begin with a unified session; then the rest of the day was broken down into one-hour blocks of four to five presentations on differing topics. Every presentation offered a unique learning experience, from topics in which I was professionally interested to those discussing a utility or program I did not yet know I needed. In the first day alone, I learned some of the intricacies of VXLAN, the hidden complications of "Hello, World," and puzzled over concerns with the potential monoculture arising within the Tor Project, to name a few. In addition to the presentations, vendors had set up tables in the main hallway, demonstrating products or giving out freebies like usb drives or pens. Once the presentations concluded for the day, attendees had opportunities to socialize and explore the city, or continue working on projects and documentation in the now-empty classrooms. These sessions would often continue late into the evening.

The final day of the conference was similar to the first. A full day of fascinating presentations concluded with a party at the Lowertown Brewery. I attended a very informative discussion of processes to streamline fuzz testing, then listened to a description of the beginning days of the FreeBSD Project, as told by Rodney Grimes, one of the original founders of the project. After lunch, a power-user's frank discussion of the limitations (in a collegiate lab) of the various BSD projects was my food for thought, followed by a demonstration of new tooling to assist release engineering with FreeBSD for dessert. Unfortunately, due to a very early flight the next morning, I had to return to the hotel and prepare for the trip home without attending the final sessions. Hopefully, I will have the opportunity to attend next year and rectify that mistake.

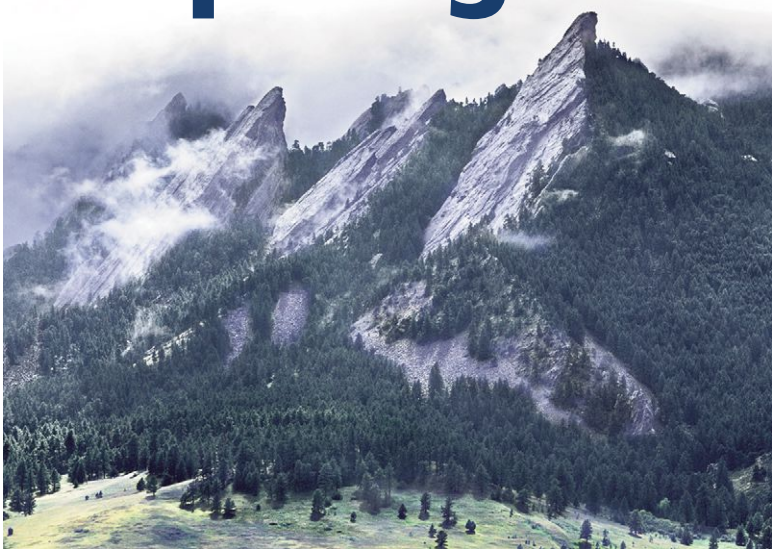
Final Thoughts

If you are active with the FreeBSD community, curious about the world of open-source computing, or just want to see exactly how many programmers it takes to change a lightbulb, BSDCan is for you. It is attended by friendly, knowledgeable people with a dizzying variety of backgrounds, all looking to both teach and learn something new. The University of Ottawa is a fine venue, with an abundance of space and plugins for several hundred people with laptops, and the city of Ottawa provided a very low-stress backdrop to the busyness of the conference. Going to BSDCan was an engaging, informative experience that I heartily recommend. If what I described sounds like paradise, then you really should plan on attending.

TIM MOORE is a technical writer with iX Systems, documenting their open-source projects like Lumina and PC-BSD. A graduate of American Military University, he holds a Master's Degree in Intelligence Analysis. Living and working in Maryville, Tennessee, he enjoys writing short stories, observing human events, and tinkering with open-source software.

Foundation Spotlight

By Deb Goodkin



I love living here in Boulder, Colorado. We're based at the foothills of the Rocky Mountains, and every day (ok, almost every day) there are many opportunities to be active outside. My typical workday includes working many hours to run the Foundation and support the Project, but it always includes going on a long run or bike ride. It's a great opportunity to take a break from my computer, get in a good workout, and process Foundation work in my head.

Sometimes, getting outside helps me take a broader view of the work. On a recent run, while I was trying to distract my brain from the heat, I started thinking about some recent emails I received from our people in the FreeBSD community. We receive a lot of emails from people asking for help from us—it might be a request for some software development work, or managing something for the Project. Some of the messaging have made me realize there are misconceptions on how big/small we are, and what we are able to do.

I thought I'd take this opportunity to tell you who we are, what we do, highlight our activities that have recently helped FreeBSD, and talk about our path going forward.

Our Roots

Back in 1999, Walnut Creek CDROM wanted to transfer the FreeBSD trademark ownership to the FreeBSD Project. However, the Project was made up of individuals, and wasn't a legal entity. Justin Gibbs, who was on the core team at the time, began researching ways they could legally own the trademark. He found that creating a non-profit would not only allow ownership of the trademark, but also ownership and protection of all the FreeBSD intellectual property. So in early 2001 he founded the FreeBSD Foundation.

For our first five years, we were run by the minimum required number of volunteers to make up our board of directors—three people. Since we didn't have a lot of funding and everyone had day jobs, we focused on a few directed funded projects, supported emerging BSD conferences, and engaged in license agreements with companies like Sun to provide FreeBSD Java binaries.

Over time, Justin realized the current situation wouldn't be sustainable, and in 2005, he hired me to run the organization. We grew our board, and I became the board Treasurer/ Secretary. We were all board members, so when people wanted to contact us, they just emailed board@freebsdfoundation.org (Note: This email address is no longer supported!). The Foundation and the board were basically synonymous.

Foundation Grows!

Fast forward to 2016. We now have six staff members and seven volunteer board members. The board helps set our strategic direction, engage with our constituents, fundraise, and has fiduciary responsibilities. The staff is responsible for implementing our strategic plans, actively running our organization, and supporting the Project.

Who Are We?

We are a team of six staff members located around the world. Our headquarters are based in

Boulder. You can read our bios here.

Let me introduce you to each one of us, so you can become familiar with the work we do, and know who to go to when you have questions or need support.

I (**Deb Goodkin**) joined the Foundation in 2005 as the Executive Director, and have full responsibility for the Foundation. My responsibilities include overseeing and guiding all aspects of the Foundation, such as software development projects, fundraising and donor engagement, budget and finances, staff management, and developing our vision and strategic plans with staff and board members.

Ed Maste joined the Foundation as our Project Development Director in 2013. He also served on the board of directors for two years prior to that. He is responsible for managing our software developer team, overseeing outside funded projects, and helping to guide our technical direction, including features, platforms, and which functionality we should support.

Anne Dickison joined the Foundation as our Marketing Director in 2014. She is responsible for marketing and advocacy for FreeBSD and the Foundation. She does a lot more than that, including heading up the Code of Conduct efforts for the Project. Not only is she producing tons of high-quality FreeBSD literature to promote and teach people about FreeBSD, she also works on community engagement by helping to get FreeBSD representation at conferences around the world.

Glen Barber joined the Foundation as our Systems Administrator in 2013, and continues in his roles as release engineering team lead, and cluster administration team member.

Kostik Belousov joined the Foundation in 2013 as a software developer. He had a long history of maintaining and improving critical kernel subsystems in FreeBSD, and as a Foundation employee continues to work on different projects, adding features, functionality, and bug fixes in FreeBSD.

Sabine Percarpio joined the Foundation as our Administration Manager in 2016. She handles donation processing, overseeing accounting, managing our office, assisting with human resources and employee benefits, managing travel grants, and is most likely the first person to respond to your call or email to the Foundation.

That's the team! We are passionate about supporting FreeBSD, but we are limited in how much support we can provide with such a small staff. The requests we are receiving, such as asking to fill more holes in Project, are telling us that we need to scale up to support these efforts. However, it can only happen if we get the donations to support this growth.

Getting lots of work done at our Foundation headquarters staff retreat in Boulder, CO. Staff members (Left to R) Anne Dickison, Ed Maste, Deb Goodkin, Glen Barber, and Sabine Percarpio.

What Are We Doing to Help FreeBSD?

Our mission is to support the FreeBSD Project and community worldwide. As you can imagine, this covers a lot of areas. Therefore, we focus on what support we can provide, based on the funding we have, to both grow the community and continue to make FreeBSD a stable, secure, innovative, and reliable operating system. Here are some highlights of what we did to help FreeBSD over the past couple of months.

Fundraising Efforts

First, I want to point out that our work is 100% funded by your donations. As of June 30, we've raised $257,570 and spent $459,900, with a 2016 fundraising goal of $1,250,000. Our Q1-Q2 financial reports will be posted in early August, so you'll be able to see how we are spending our money. For us to continue this level of support, we need your donations. Please consider making a donation here.

OS Improvements

The Foundation improves FreeBSD by funding software development projects approved through our proposal submission process and our internal software developer staff members. We recently completed a project to improve the stability of the vnet network stack virtualization infrastructure. Another project, still in progress, is bringing new functionality, and stability and performance enhancements, to the 64-bit ARM port of FreeBSD. We are also sponsoring work to bring blacklistd to FreeBSD, a practical approach to con-

nection-based security control.

In addition, on an ongoing basis, our software developers make many improvements, implement new features, and fix bugs to keep FreeBSD reliable and stable. Here is a list of some of the work done over the past few months:

- Implemented robust mutexes support, as part of ongoing efforts to bring our threading library into POSIX compliance and feature completeness.
- Documented kernel interfaces used by the threading library and produced almost 30 pages of technical text.
- Completed and committed the elimination of the `pvh_global_lock` from the `amd64 pmap`, which removed a bottleneck of a highly contested lock.
- Added filesystem thoughput resource control limits (RCTL).
- Committed iSER initiator support.
- Added support for rerooting into NFS.
- Added `iscsictl`, which makes it possible to enable and disable iSCSI sessions.
- Investigated the state of reproducible builds in the ports tree, with work in progress to address identified issues.
- Updated ELF Tool Chain with bug fixes and improved handling of malformed input.
- Investigated using lld, the linker from the LLVM family, to link the FreeBSD base system. Investigated and tested fixes for the issues found.
- Managed the arm64 development project and investigated and fixed a number of bugs.
- Imported llvm libunwind and prepared it for use in FreeBSD.
- Investigated and reviewed the blacklistd proposal and patches.
- Facilitated biweekly calls for network transport, DTrace, and the graphics stack to help coordinate efforts in these areas.

Release Engineering

As I mentioned earlier, the Foundation employs a full-time person, Glen Barber, to work on release engineering to ensure releases are reliable, stable, and on time. Last quarter, Glen helped with the 10.3-RELEASE and starting the 11.0-RELEASE cycle.

Getting Started with FreeBSD Project

We kicked off a new Getting Started with FreeBSD Project. The purpose of the project is to develop how-tos and getting-started guides and videos for new people interested in trying out FreeBSD for the first time. While promoting FreeBSD at events, we found many people interested in trying out FreeBSD, so we decided as part of our education initiative that we should provide easy-to-follow tutorials to get these people on board quickly!

For this project, we brought on an intern with no FreeBSD, Linux, or any command line operating system experience, to figure out on his own how to install and use FreeBSD. We did this to get a new person's perspective on using the current documentation and what made learning about FreeBSD and installing it on his desktop easy/difficult. With occasional help from members of the community, he is writing easy-to-follow how-to guides to make it easier for new users to get started. These will also be perfect guides to use at hackathons to help beginners get started with FreeBSD.

You can check out our how-to guides here.

FreeBSD Advocacy and Education

A large part of our efforts are dedicated to advocating for the Project. We do this in a number of ways, including promoting work being done by others with FreeBSD, and producing advocacy literature to teach people about FreeBSD while helping to make the path to using FreeBSD or contributing to the Project easier. We also attend FreeBSD and non-FreeBSD events, and work with other FreeBSD contributors to volunteer to run FreeBSD events, staff FreeBSD tables, and give FreeBSD presentations.

Some of our recent accomplishments include:

- Creating a FreeBSD page on our website to promote FreeBSD derivative projects and showcase FreeBSD users.
- Promoting FreeBSD research by creating a research page on our site and conference handout.
- Creating guidelines and a repository for using Project and Foundation logos.
- Showcasing FreeBSD contributors by publishing two new Faces of FreeBSD stories about Michael Lucas and Kris Moore.
- We publish this magazine, which is another platform for providing informative and interesting articles about FreeBSD.
- On the FreeBSD education front, George Neville-Neil and Robert Watson continued teaching and developing open-source FreeBSD teaching materials at teachbsd.org.

Microsoft Loves FreeBSD!

We've worked with Microsoft over the last two years, to get FreeBSD in the Azure Marketplace. Microsoft will maintain their own FreeBSD images, and provide FreeBSD support to their customer base. This is an exciting partnership between the FreeBSD Project and Microsoft. You can read more about FreeBSD on Azure here.

Conferences and Events

The FreeBSD Foundation sponsors many conferences, events, and summits around the globe. These events can be BSD-related, open-source, or technology events geared toward underrepresented groups.

We support the FreeBSD-focused events to help provide a venue for sharing knowledge, to work together on projects, and facilitate collaboration between developers and commercial users. This all helps provide a healthy open-source ecosystem. We support the non-FreeBSD events to promote and raise awareness about FreeBSD, to increase the use of FreeBSD in different applications, and to recruit more contributors to the Project. Lastly, we provide travel grants to FreeBSD contributors to have face-to- face opportunities to share knowledge, work on projects, and to learn more about FreeBSD by attending talks, presentations, and tutorials.

In April, board member Benedict Reuschling helped organize and run a hackathon in Essen, Germany, April 22–24. He then attended the Open Source Datacenter conference in Berlin, with FreeBSD contributor Allan Jude, to speak about "Interesting things you can do with ZFS," which highlighted OpenZFS features and how well they work on FreeBSD.

During the last quarter alone, we promoted FreeBSD at:

- Flourish: April 1–2 in Chicago, Illinios.
- LFNW: April 23–24 in Bellingham, Washington.
- OSCON: May 18–19 in Austin, Texas.
- USENIX ATC: June 22–23 in Denver, Colorado.
- Deb Goodkin and Dru Lavigne attended the Community Leadership Summit, May 14–15 in Austin, Texas.
- Our team attended BSDCan and the Ottawa Developer Summit. We held our annual board meeting to vote on officers, board members, and work on our strategic planning. Most of us attended and participated in the developer/vendor summits. Board member Kirk McKusick presented "A Brief History of the BSD Fast Filesystem."
- Ed Maste gave a presentation on Reproducible Builds in FreeBSD. Board member George Neville-Neil helped run the vendor summit and gave a talk called "Through the Wire."
- We provided travel grants to five FreeBSD contributors to attend BSDCan.

Legal/FreeBSD IP

The Foundation owns the FreeBSD trademarks, and it is our responsibility to protect them. We continued to review requests and grant permission to use the trademarks. We also provided legal help for questions the core team had about specific patents.

FreeBSD Community Engagement

We launched our first Community Survey. The purpose was to get input to help us determine our direction and how we should support the Project. Thank you to everyone who provided feedback!

Code of Conduct—Anne Dickison, our Marketing Director, has been overseeing the efforts to rewrite the Project's Code of Conduct to help make this a safe, inclusive, and welcoming community.

Going Forward!

We accomplished a lot these past few months, and we plan on increasing our support. We've held two all-day board meetings where we spent time working on our strategic plan and identifying areas we believe we need to support to keep FreeBSD relevant and sustainable.

Over the next year or two, you'll see the Foundation helping with security and improving developer tools. We'll be reaching out to more commercial users to help us fund these efforts.

We Need Your Help!

We are a small, yet mighty, organization. We are passionate about what we do and work hard to support FreeBSD, but we can't do this work without your help.
Please consider making a donation so we can continue and increase our support to make FreeBSD the best operating system available!
https://www.freebsdfoundation.org/donate/ •

DEB GOODKIN is the Executive Director of the FreeBSD Foundation. She's thrilled to be in her 11th year at the Foundation and is proud of her hardworking and dedicated team. She spent over 20 years in the data storage industry in engineering development, technical sales, and technical marketing. When not working, you'll find her on her road or mountain bike, running, hiking with her dogs, skiing the slopes of Colorado, or reading a good book.

SEE TEXT ONLY

new faces of FreeBSD

BY DRU LAVIGNE

Regular readers of this column will note that, starting with this issue, the focus has changed somewhat. FreeBSD has a long history of providing mentorship as part of its commit bit process and the Project's success is due, in part, to its ability to continually attract new contributors. This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community.

This month, the spotlight is on Carlos Jacobo Puga Medina (left) and Tobias Berner (right) who both became ports committers in July 2016.

Tell us a bit about yourself, your background, and your interests.

Carlos: My name is Carlos Jacobo Puga Medina and I was born in Jaen in the South of Spain. Currently, I'm learning more about FreeBSD and I'm doing this since I stopped working in the Army. From a very young age I liked new technologies and everything that surrounds them.

My biggest interests are coding, writing, languages, and trekking. I enjoy learning new things.

Tobias: My name is Tobias Berner and I'm a 31-year-old math student from Switzerland. My interests, apart from the obvious, are playing video games, hiking, cycling, cooking, and basically enjoying life.

How did you first learn about FreeBSD and what about FreeBSD interested you?

Carlos: Well, my history with FreeBSD started around 2011 when I was still working in the Army. I was looking for a new operating system and decided to try FreeBSD. I was fascinated by how it works, who benefits, and how it is developed. At that time, I discovered how friendly the FreeBSD community is, largely thanks to the FreeBSD Forums, and especially to Warren Block (wblock) for his unconditional support, valuable advice, and guidance.

From then until now, I can safely admit that FreeBSD is my daily driver.

Tobias: I first learned about FreeBSD when I was experimenting with operating systems on my first computer during middle school. I wanted to try this "Linux" thing, in parallel to Windows. After playing with SUSE and some other distros, I somehow, somewhere stumbled over the Beastie Logo, which I found kind of cute. So I gave FreeBSD a try, and that's how I came to install FreeBSD 5. I didn't really have a clue what I was doing, but somehow it seemed to work, most of the time.

And then I began to love how software was installed using the ports tree. It's the one thing that made me stick with FreeBSD over the various Linux distros even in the dark times, when I got a core2d and there was no 64bit nVidia driver.

How did you end up becoming a ports committer and which ports are you responsible for?

Carlos: After reading the ports documentation and following the mailing lists, I started to port some software and to fix some ports, mainly unmaintained ones.

A few months ago, one of my current mentors, Jason Unovitch (junovitch), asked if I

wanted a commit bit. The answer came pretty soon. Right now, I'm maintaining some ports, as I did before: devel/boehm-gc, net-im/corebird, security/libgcrypt, and multimedia/mpv, among others.
Tobias: As a KDE user myself, I wanted to try the new Plasma5 desktop. However, there were no ports for it, so I started to create them. After some time, I got the rights to mess around in the kde@ development repository, and put my Plasma5 ports in there. I also started to help updating and testing the other ports maintained by kde@. Which then, after some time, led to me becoming a committer.

The ports I try to help with and keep up-to-date are all the ones maintained by kde@. This includes KDE, Qt, PyQt, digikam, calligra, and so on.

How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a ports committer?

Carlos: My experience has been nice because I'm self-motivated and enthusiastic and take great pride in my tasks and roles. I started to enjoy the experience more after I became more involved in the FreeBSD Project. I think that enabling new people to contribute to the Project is part of the success we are having. In fact, I'm learning new things to enhance my skills and maximize my effort on things that bring a real value to the Project. Also, I'm really looking forward to attending some upcoming BSD conferences and to personally meet my fellow FreeBSD community members.

A good starting point is the FreeBSD "Becoming A Committer" wiki page (https://wiki.freebsd.org/BecomingACommitter). Here are some steps that work:

- Read the FreeBSD documentation and related blogs.
- Subscribe to the FreeBSD mailing lists and join the FreeBSD Forums.
- Submit fixes to bugs via bugzilla.
- Go to BSD conferences and trade shows.
- Contribute!

The FreeBSD Committers Guide (https://www.freebsd.org/doc/en/articles/committers-guide/) discusses how the Project works, what rules to follow, and how to actually make svn commits. Anyone who is seriously interested in becoming a committer should read this to see how committers do what they do.

Tobias: It has been very interesting, and a little scary, but the people are very welcoming. And I hope the blood pressure will go down as the number of commits rises. As for advice, I can only offer to stick with it and try to contribute, even if it takes some time until you get there.

DRU LAVIGNE is a Director of the FreeBSD Foundation and Chair of the BSD Certification Group.

THROUGH OCTOBER 2016 BY DRU LAVIGNE

2016 Events Calendar

The following BSD-related conferences will take place in September and October 2016. More information about these events, as well as local user group meetings, can be found at **www.bsdevents.org**.

EuroBSDCon • Sept 22–25 • Belgrade, Serbia

https://2016.eurobsdcon.org/ • The premier annual European conference on BSD operating systems attracts skilled engineering professionals, software developers, computer science students and professors, and users from all over Europe and other parts of the world. The conference includes a Developer Summit, Vendor Summit, tutorials, and presentations. The BSDA certification exam will also be available during this event. Registration is required.

OpenZFS • Sept 26 & 27 • San Francisco, CA

http://open-zfs.org/wiki/OpenZFS_Developer_Summit • The fourth annual OpenZFS Developer Summit will be held in San Francisco, California. The goal of this event is to foster cross-community discussions of OpenZFS work and to make progress on some of the proposed projects. This 2-day event consists of 1 day of presentations followed by a 1-day hackathon. A nominal registration fee is required for this event and space is limited.

Ohio LinuxFest • Oct 7 & 8 • Columbus, OH

https://ohiolinux.org/ • There will be a FreeBSD booth and several FreeBSD-related talks at Ohio LinuxFest, to be held at the Hyatt Regency Columbus. The BSDA certification exam will also be available during this event. There is a nominal registration fee for this conference.

Grace Hopper • Oct 19–21 • Houston, TX

http://ghc.anitaborg.org/ • There will be a FreeBSD booth and a Getting Started with FreeBSD Workshop at the annual Grace Hopper Celebration of Women in Computing. There is a registration fee for this conference and space is limited.